CIRCUS, un générateur de composants pour le traitement des langages visuels et textuels

Jean-Yves Vion-Dury

THÈSE

présentée par

Jean-Yves Vion-Dury

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
(ARRÊTÉ MINISTÉRIEL DU 5 JUILLET 1984 ET DU 30 MARS 1992)

Spécialité :
INFORMATIQUE

CIRCUS: Un générateur de composants pour le traitement des langages visuels et textuels

Soutenue le 17 juin 1999

Jury:

Président:	Sacha Krakowiak
Directeur:	Roland Balter
Rapporteurs:	Isabelle Attali
	Hélène Kirchner
Examinateurs:	Jean-Marc Andreoli
	Didier Bert

THÈSE PRÉPARÉE AU SEIN DU PROJET SIRAC (INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, UNIVERSITÉ JOSEPH FOURIER, UNIVERSITÉ DE SAVOIE)

Table des matières

1 Introduction **9**
1.1 Contexte 11
1.2 Objectifs 12
1.3 Organisation du document 14

I État de l'art et contexte 15

2 Langages et environnements : vers le syndrôme de Babel **17**
2.1 introduction 19
2.2 L'évolution des langages de programmation 19
2.3 Les environnements de développement 22
2.4 Les langages visuels interactifs 24
2.5 conclusion 34

3 Théorie et pratique des outils de génération **37**
3.1 Introduction 39
3.2 La théorie : au cœur de l'évolution des langages 39
3.2.1 Introduction 39
3.2.2 Traitements lexicaux 40
3.2.3 Traitements syntaxiques 43
3.2.4 Traitement des types 51
3.2.5 Traitements sémantiques 53
3.3 Outils de génération 62
3.3.1 Progress : un langage et environnement de spécification par réécriture de graphes 62
3.3.2 Optimix : réécriture de graphes pour l'optimisation de code 64
3.3.3 Centaure : environnements intégrés et sémantique naturelle 66
3.4 Conclusion 68

II Le Langage Circus 71

4 Le noyau du langage **75**

4.1 Syntaxe . . . 77
4.1.1 noyau fonctionnel . . . 77
4.1.2 définitions modulaires . . . 77
4.2 Sémantique opérationnelle . . . 78
4.2.1 Introduction . . . 78
4.2.2 Environnement d'exécution . . . 78
4.2.3 Le système de transition : calcul des termes . . . 79
4.2.4 Le système de transition : calcul des types . . . 86
4.3 Système de types . . . 87
4.3.1 Environnement de typage et jugements . . . 87
4.3.2 Types bien formés . . . 87
4.3.3 Termes correctement typés . . . 88
4.3.4 Sous-typage . . . 90
4.3.5 Type minimal d'un terme . . . 91
4.3.6 Equivalences de types . . . 94
4.3.7 Opérations sur les types . . . 95
4.3.8 Opérateurs sur les termes . . . 102
4.4 Correction du système de types . . . 107
4.4.1 Environnement d'exécution correctement typé et conforme . . . 107
4.4.2 Complétude du système de transition . . . 107
4.4.3 Préservation du type lors des réductions . . . 108
4.4.4 Préservation structurelle d'un type lors de ses dérivations . . . 108
4.4.5 Correction globale du système de types . . . 110
4.5 Interprétation . . . 111
4.5.1 Calcul des dérivations élémentaires . . . 111
4.5.2 Calcul de Fermeture . . . 114
4.5.3 Evaluation des types . . . 115
4.5.4 Interprétation complète . . . 116
4.5.5 Propriétés . . . 116
4.6 Contrôle des types . . . 118
4.6.1 Validation de confluence . . . 118
4.6.2 Création d'un environnement d'exécution conforme . . . 119
4.6.3 Types bien formés . . . 120
4.6.4 Sous-typage . . . 121
4.6.5 Union et intersection de types . . . 121
4.6.6 Inférence de type . . . 122
4.6.7 Conformité de types . . . 126
4.6.8 Propriétés . . . 126
4.7 Synthèse . . . 128

5 Actions impératives et machines abstraites polymorphes 129
5.1 Introduction . . . 131
5.2 Actions impératives . . . 132
5.2.1 Syntaxe . . . 132
5.2.2 Typage . . . 133
5.2.3 Sémantique . . . 134

5.2.4 Correction du système de types . . . 134
5.3 Filtrage . . . 138
5.3.1 Syntaxe . . . 138
5.3.2 Règles de typage . . . 138
5.3.3 Sémantique . . . 140
5.3.4 Correction du système de types . . . 141
5.4 Coordination . . . 146
5.4.1 Syntaxe . . . 146
5.4.2 Règles de typage . . . 146
5.4.3 Sémantique . . . 147
5.4.4 Correction du système de types . . . 148
5.5 Concurrence . . . 150
5.5.1 syntaxe . . . 150
5.5.2 typage . . . 150
5.5.3 sémantique . . . 150
5.5.4 Correction du système de types . . . 151
5.6 Système d'actions . . . 152
5.6.1 Syntaxe . . . 152
5.6.2 Sémantique . . . 152
5.6.3 Typage . . . 152
5.6.4 Correction du système de types . . . 153
5.7 Machines abstraites . . . 155
5.7.1 Syntaxe . . . 155
5.7.2 Typage . . . 155
5.7.3 Sémantique . . . 156
5.7.4 Correction du système de types . . . 157
5.8 Synthèse . . . 159

6 Types de données structurés 161
6.1 Introduction . . . 163
6.2 Préliminaires . . . 163
6.2.1 Types structurés et énumérations . . . 163
6.2.2 Union de types . . . 164
6.2.3 Types principaux . . . 164
6.2.4 Calcul du type principal - calcul du type d'une opération de filtrage . . . 165
6.3 Structures . . . 166
6.3.1 Syntaxe . . . 166
6.3.2 Règles de typage . . . 166
6.3.3 Union minimale et intersection maximale . . . 167
6.3.4 filtrage de structures . . . 168
6.3.5 Sémantique . . . 169
6.3.6 Correction du système de type . . . 171
6.3.7 Contrôle de type . . . 172
6.4 Sequences . . . 175
6.4.1 Règles de typage . . . 175
6.4.2 Union minimale et intersection maximale . . . 175

6.4.3 filtrage des séquences . . . 176
6.4.4 Sémantique . . . 177
6.4.5 Correction du système de type . . . 178
6.4.6 Contrôle de type . . . 179
6.5 Ensembles . . . 182
6.5.1 Syntaxe . . . 182
6.5.2 Sémantique . . . 182
6.6 Tuples . . . 184
6.6.1 Règles de typage . . . 184
6.6.2 Union minimale et intersection maximale . . . 184
6.6.3 filtrage de tuples . . . 185
6.6.4 Sémantique . . . 186
6.6.5 Correction du système de type . . . 187
6.6.6 Contrôle de type . . . 187
6.7 Dictionnaires . . . 190
6.7.1 Règles de typage . . . 190
6.7.2 Union minimale et intersection maximale . . . 191
6.7.3 filtrage des dictionnaires . . . 191
6.7.4 Sémantique . . . 192
6.7.5 Correction du système de type . . . 194
6.7.6 Contrôle de type . . . 195
6.8 Synthèse . . . 198

7 Compositions syntaxiques 199
7.1 Introduction . . . 201
7.2 Actions impératives . . . 203
7.2.1 Syntaxe . . . 203
7.2.2 Sémantique . . . 203
7.2.3 Typage . . . 203
7.3 Machines abstraites . . . 205
7.3.1 Syntaxe . . . 205
7.3.2 Sémantique . . . 205
7.3.3 Typage . . . 208
7.3.4 Correction . . . 208
7.4 Synthèse . . . 209

III Architectures transformationnelles et interactives 211

8 Machines visuelles 219
8.1 Introduction : représentation et interaction . . . 221
8.2 Types visuels . . . 221
8.2.1 Attributs perceptuels . . . 222
8.2.2 Glyphes élémentaires . . . 223
8.2.3 Glyphes composites . . . 224
8.3 Syntaxe de l'interaction . . . 224

8.4 Syntaxe visuelle 228
8.5 Instructions graphiques 229

9 Le langage *While* décrit en *Circus* : version textuelle 237
9.1 Analyse syntaxique 239
9.2 Convertisseur en syntaxe abstraite 240
9.3 Contrôleur de types 241
9.4 Interpréteur 243
9.5 Générateur de code 245

10 Le langage *While* décrit en *Circus* : version visuelle et interactive 249
10.1 Présentation d'ensemble 251
10.1.1 Spécification de la structure du programme 251
10.1.2 Spécification des opérations de transformation 252
10.2 Syntaxe de l'interaction 252
10.2.1 Spécification de la structure du programme 252
10.2.2 Spécification des opérations de transformation 253
10.3 Syntaxe visuelle 253
10.3.1 représentation structurelle 253
10.3.2 Spécification des opérations de transformation 255
10.4 Couplage avec les noyaux de traitement 255
10.4.1 contrôle des types 257
10.4.2 génération 257
10.5 Synthèse 258

11 Conclusion 259
11.1 Récapitulatif 261
11.2 Étude du langage 261
11.3 Expérimentation 263
11.4 Perspectives et enjeux 263

Bibliographie 267

Annexes 277

A Demonstrations 277

Introduction

1.1 Contexte

Cette thèse a débuté alors que l'action de recherche *Sirac* [1] était lancée dans le cadre de la naissance de l'*Inria Rhônes-Alpes* en 1995, autour de la double thématique des systèmes distribués (*Arias*) et outils pour applications coopératives (*Olan*). Parallèlement, le centre de recherche *Xerox* [2] développait un axe de recherche orienté vers les *technologies de la Coordination*, et desirait accueillir un doctorant de l'université, afin de concrétiser une collaboration scientifique. Une partie de la tâche préliminaire à cette thèse a donc été d'identifier l'intersection des domaines de compétence, des interêts communs ou divergents des différentes équipes de recherche.
J'avais été conduit auparavant, dans le cadre de mon année d'ingéniorat *CNAM* [116], puis de mon *DEA*[117] effectués alors au Laboratoire Bull-IMAG/Systèmes, à étudier et réaliser des outils de mise au point pour applications à objets distribués, dont la particularité résidait dans la mise en oeuvre de techniques de visualisation interactive.
Cette approche avait alors largement convaincu de l'importance à venir des technologies visuelles, dès lors qu'elles étaient bien intégrées aux applications. Elle avait également suggéré de nombreuses voies permettant d'automatiser partiellement le développement de telles applications, mais aussi, de compenser les nombreuses limites de l'approche purement "interactive" des tâches de spécification pouvant être associées à cette approche.
Dans le même temps, les orientations du projet *Sirac* se dessinaient en direction d'une plateforme de configuration d'applications distribuées inspirée de l'approche *Darwin* [77], reposant sur un formalisme graphique, et faisant apparaître de nombreux besoins en termes d'environnements de développement et de langages. Ces mêmes besoins apparaissaient au centre de recherche Xerox, en rapport avec la plateforme et le langage de coordination *CLF* développés dans l'équipe *Coordination Technologies*. Il apparut alors que le travail de thèse pourrait se focaliser sur l'aide à la génération d'environnements de développement présentant de telles caractéristiques, c'est à dire composés d'outils visuels (édition, mise au point, surveillance à l'exécution) fortement couplés aux environnements d'exécution et aux outils de compilation. Dans cette perspective, les deux projets sont tout naturellement devenus des cibles d'expérimentation et d'application. Les besoins de chacun des environnements *Olan* et *CLF*, ainsi que les différentes cultures scientifiques développés dans ces projets, se combinèrent pour constituer le cadre de ce travail de recherche. *Olan*, de par sa vocation de langage de description d'architecture et de configuration, se prêtait particulierement bien à la spécification visuelle intéractive, tout en offrant différents niveaux d'abstraction, dont certains étaient mieux traités au travers d'approches textuelles "classiques". Ainsi, le besoin d'une approche mixte "visuelle-textuelle" de l'environnement de développement, dans lesquels les noyaux de traitement pouvaient être réutilisés, partagés ou tout simplement mis en communication, était clairement identifié. *CLF* pouvait naturellement bénéficier de ces qualités, tout en favorisant une approche basée sur la richesse des structures de données : les langages et outils de cet environnement devaient modéliser des flots de données et des des schémas de communication riches et variés.

[1] Systèmes Informatiques Répartis pour Applications Coopératives. *Sirac* est un projet de l'Institut de Mathématiques Appliquées de Grenoble, de L'Institut National de Recherche en Informatique et Automatique, de l'Institut National Polytechnique de Grenoble, de l'Université Joseph Fourier et enfin de l'Université de Savoie

[2] installé à Meylan et inauguré en 1994, il fut d'abord appellé *Rank Xerox Research Centre*, puis renommé *Xerox Research Centre Europe* en 1997

1.2 Objectifs

Le langage Circus, objet de cette étude, est destiné aux concepteurs de langages et environnements de développement associés. Il vise à faire évoluer les outils de génération actuels, qui, au niveau du monde industriel, se limitent à l'analyse lexicale et syntaxique, et dont le prototype est représenté par le couple *Lex/Yacc*. Si aujourd'hui, les offres publiques ou industrielles les plus récentes, comme *Pccts*/*Antlr*, [90] proposent des générateurs d'analyseurs syntaxiques riches et plus souples que les outils d'origine, l'effort reste localisé au niveau frontal (lexical et syntaxique), et ne peut pas prétendre à une généralisation de la transformation des structures aux autres niveaux de traitement ([89]). En effet, des aspects essentiels du traitement des langages, tels que les transformations intermédiaires, les vérifications sémantiques, le contrôle des types, l'optimisation, la génération de code exécutable, ou de code source vers un langage cible ne bénéficient ni d'une description de haut niveau comparable aux grammaires formelles décrivant des syntaxes, ni des procédés de génération automatique pouvant y être associés. Deux questions fondamentales se posent alors dans ce contexte : est-il possible de définir un formalisme capable de prendre en compte ces différents niveaux du traitement linguistique ? Ce formalisme peut-il être suffisament expressif et simple tout en offrant des possibilités réalistes au niveau de l'implantation logicielle ?

Outre ces deux interrogations fondamentales, il apparaît également nécessaire de définir quels sont les langages visés ainsi que les environnements susceptibles de leur être associés. Les langages pouvant être générés ou partiellement générés à partir de descriptions *Circus* sont soit textuels "classiques", soit visuels, soit mixtes c'est à dire susceptibles de partager des noyaux de traitements communs. Cette approche est ambitieuse, mais aussi prometteuse, car elle suppose la mise en évidence de déterminants fondamentaux dans les diverses transformations liées aux traitements. De plus, il est nécessaire de prendre en compte les exigences liées aux langages "modernes" telles que les contrôles de types sophistiqués, la génération vers des cibles multiples, ou une optimisation de qualité. Plus spécifiquement, l'avènement des technologies de communication actuelles favorise l'émergence d'une nouvelle classe de langages et d'environnements orientés vers l'intégration de sources de différentes natures, l'inter-opérabilité, et la répartition sur des plateformes hétérogènes.

Les langages visuels, quant à eux, engendrent de multiples difficultés liées essentiellement à la richesse du système de représentation. Celui-ci est considéré comme multidimensionnel, en opposition au système linéaire et unidimensionnel que sont les simples textes. Ainsi, dans un langage visuel, des paramètres tels que la couleur, la forme et la position des objets graphiques sont conjointement utilisés dans la définition de la syntaxe et de la sémantique. Dans un langage textuel, seul l'ordre des mots et leur nature sont utilisés.

Toutefois, il ne convient pas d'opposer ces deux approches du langage, fortement complémentaires, leurs qualités et leurs défauts se situant à des niveaux différents. Si les langages textuels bénéficient de cinquante années de recherches et de développements, l'étude des langages visuels est beaucoup plus récente dans la mesure où ils sont dépendants des performances graphiques des ordinateurs. Et pourtant, les avancées récentes réalisées dans les formalismes grammaticaux ainsi que dans l'analyse syntaxique multidimensionnelle, permettent d'entrevoir un moyen d'unifier les techniques issues de ces deux domaines dans un cadre commun. *Circus* peut être considéré comme une contribution susceptible de favoriser et de dynamiser cette évolution devenue indispensable, dès lors qu'elle peut améliorer significativement la puissance et la qualité des outils linguistiques de demain.

Les environnements associés à de tels langages doivent vérifier des caractéristiques qui paraissent à première vue antagonistes : d'une part, présenter de bonnes capacités d'extensibilité, d'autre part, rester fortement couplés aux noyaux de traitement du langage considéré. L'extensibilité permet, par exemple, de rajouter à un couple de base constitué d'un éditeur et d'un compilateur, un metteur au point. Par la

suite, en fonction des évolutions du développement, on pourrait encore adjoindre un formateur de textes sources, un décompilateur, un vérificateur statique de codes sources, un moniteur d'exécution, et bien d'autres encore. Tous ces éléments de l'environnement ont en commun certaines connaissances liées au langage traité : le formateur de textes utilise généralement les connaissances syntaxiques mises en oeuvre dans la partie frontale du compilateur, le décompilateur utilise les connaissances liées à la génération du code objet mis en oeuvre dans la partie "back-end" du compilateur. Afin de traduire cette connaissance commune par une réutilisation réelle ou effective des noyaux de traitement, il est nécessaire d'envisager une architecture particulière des outils, favorisant la réutilisation, la modularité et l'intégration des différents composants de traitement.

Ainsi, si les fonctionnalités et les propriétés des outils de traitement de langages auxquels nous nous intéressons sont importantes, leur structure interne l'est tout autant. En effet, il est légitime d'attendre les mêmes qualités d'un méta-langage que d'un simple langage : modularité des spécifications, réutilisation optimale, système de types performant et souple. De plus, les composants générés doivent particulièrement bien se prêter à la composition au sein d'architectures très diverses, comme évoqué précédemment. L'ensemble de ces objectifs de premier niveau a naturellement induit un objectif de second niveau : découvrir le formalisme central permettant de spécifier de manière uniforme et cohérente les composants de traitement. Ce formalisme devant en outre posséder à la fois une bonne expressivité tout en restant accessible à un utilisateur non spécialiste de la théorie des langages.

Ce travail de recherche s'est donc fondé sur trois principes initiaux, visant à généraliser le traitement des langages à des transformations de structures : (i) offrir un modèle de données apte à modéliser tous les niveaux de traitements impliqués dans les transformations ; (ii) offrir une abstraction simple, adaptée à la transformation de ces données et possèdant de "bonnes" propriétés compositionnelles, et (iii) prendre en compte l'architecture interne des transformations ainsi que le contrôle de l'exécution.

Un modèle de données. L'introduction d'un modèle de données peut aisément se justifier : lorsque la puissance de modélisation est adaptée aux applications visées par le langage, l'écriture des programmes est simplifiée, les performances et la clarté des spécifications s'en trouve améliorée. Toutefois, une première difficulté est d'intégrer étroitement ce modèle de données au modèle d'exécution. D'autre part, le contrôle statique des types de données et de leur utilisation permet de fiabiliser les applications mais également d'améliorer les performances à l'exécution.

Une abstraction dédiée à la transformation. Si l'analyseur syntaxique, généré à partir d'une grammaire hors-contexte, représente l'outil même de la phase frontale des traitements, il est naturel d'attendre un outil similaire pour les phases "profondes" associées aux transformations de structures. Pour répondre aux propriétés de réutilisation et de partage des noyaux de traitement, ces "agents" de transformation devront être composables, soit à l'exécution, soit au niveau du code source, lors de la spécification. Le terme "composition" signifie que des opérations simples doivent permettre de créer de nouveaux agents à partir d'agents existants. L'héritage dans les langages à objets est un exemple de composition, de niveau source. Le lancement d'agents de traitement en parallèle ou en séquence constitue un autre exemple de composition, mais au niveau de l'exécution.

Contrôle de l'exécution. Cette partie est probablement moins évidente à justifier, car l'état de l'art montre que ce facteur n'a pas été identifié comme déterminant dans la communauté de recherche organisée autour de cette thématique. La volonté de prendre en compte les aspects temporels liés aux langages visuels dynamiques, tels qu'ils peuvent être induits par des éditeurs syntaxiques intéractifs, fait pourtant

clairement surgir la nécessité d'un contrôle fin des étapes de transformations. Par exemple, lors d'une modification incrémentale d'un programme source, les étapes de calcul permettant de reconstruire en temps réel l'arbre d'analyse doivent être connues et maîtrisées. Cela permet d'éviter des calculs "en avalanche", tels qu'ils s'observent parfois dans les systèmes à base de règles, et qui peuvent induire des temps de réponse rédhibitoires pour des applications interactives. Traditionnellement, en considérant une échelle d'abstraction croissante dans les spécifications, les hypothèses d'exécutions associées vont du totalement explicites (langages impératifs), au totalement implicites (stratégies d'application des règles dans un système de réécriture). Plus précisément, dans les langages à haut-niveau d'abstraction, reprendre le contrôle de l'exécution demande une expertise très élevée, produit du code impur et va à contre-sens du niveau d'abstraction (typiquement, tenter de maîtriser les étapes d'exploration et retours arrière dans un programme *Prolog*). Or, le niveau intermédiaire dans l'échelle d'abstraction est extrêmement intéressant, car non seulement il ouvre la voie à des comportements au déterminisme modulable, mais en plus, il autorise des schémas de compositions plus fins, capables de maîtriser et propager ces caractéristiques opérationnelles.

Ces trois principes ont été unifiés au travers d'une étude théorique et expérimentale, puis concrétisés dans un langage original, spécialisé pour traiter la transformation de structures.

1.3 Organisation du document

Ce document présente le langage Circus, ainsi que le travail de recherche qui a conduit à sa réalisation. L'étude de l'existant, l'état de l'art dans les domaines concernés , la réflexion scientifique ayant motivé la démarche sont présentés explicitement dans les deux premières sections. Toutefois, un effort particulier sera rendu afin de les faire apparaître tout au long des développement suivants, lorsque leur évocation sera susceptible d'éclairer le lecteur sur les choix ou les éléments sous-jacents déterminants dans cette étude.

La première partie est constituée de deux chapitres. Le premier propose une analyse des grands axes impliqués dans l'élaboration de ce travail et s'efforce de mettre en évidence les tendances passées, présentes et futures de cette évolution. La vision qui s'en dégage est largement reliée aux choix scientifiques envisagés par la suite. Le chapitre suivant étudie plus précisément les technologies d'aide à la réalisation d'outils de traitement de langages, à la fois dans leur évolution progressive et dans leur aboutissement actuel. Il permet ainsi d'introduire les bases nécessaires à la compréhension de ce travail, et également de le positionner par rapport aux travaux existants.

La seconde partie présente l'étude théorique des principaux aspects du langage, en faisant porter un effort particulier sur le système de types et les *machines abstraites polymorphes*, qui constituent la contribution essentielle de ce travail.

La partie 3 présente la mise en œuvre du langage, de manière à illustrer de manière plus précise et concrète les concepts théoriques développés dans la partie précédente. En développant un langage d'étude textuel et visuel, il sera montré de quelle façon le générateur *Circus* peut être utilisé, parfois de manière précise et détaillée, parfois de manière plus synthétique.

Le document conclut sur une mise en perspective de *Circus* et de ses prolongements futurs.

Première partie

État de l’art et contexte

Langages et environnements : vers le syndrôme de Babel

2.1 introduction

Ce chapitre analyse l'évolution de trois axes distincts (les langages de programmation, les environements de developpement et les langages visuels interactifs), en s'efforçant de décrire à la fois l'aspect "externe" de ces applications et leur aspect "interne". Nous entendons par "externe", les fonctionnalités, concepts et propriétés perçus et manipulés par les utilisateurs. L'aspect "interne" concerne les concepts, outils théoriques et pratiques mis en œuvre par les concepteurs. L'axe "langages visuels" étant de loin le moins connu, il sera plus extensivement développé que les axes "langage de programmation" et " environnements de developpement". Ce chapitre ne vise absolument pas à établir une analyse exhaustive de domaines d'études aussi vastes. Au moyen d'un choix de travaux et réalisations considérés comme significatifs des différentes étapes de l'évolution, il propose d'identifier les facteurs fondamentaux susceptibles de continuer à agir et d'influer sur les concepts, connaissances et technologies de demain.

2.2 L'évolution des langages de programmation

Le niveau zéro dans l'échelle des langages est représenté par le langage machine c'est à dire des séquences d'instructions capables d'être interprétées par un processeur physique [1].

Dès lors, l'évolution des langages peut être vue comme l'introduction progressive d'abstractions susceptibles de faciliter la tâche du programmeur, d'améliorer sa productivité, tout en conservant au maximum l'expressivité et la puissance de traitement disponibles au niveau zéro. Ces abstractions sont typiquement des facilités mnémoniques telles que la représentation d'adresses physiques, de nature numérique, par des adresses symboliques, de nature textuelle. C'est ce qu'ont amené les langages d'assemblage, dont les premiers représentants ont vu le jour dans les années 1940, afin de répondre aux difficultés rencontrées dans la programmation des calculateurs, qui se faisait alors en code binaire pur, voir même par cablage direct. Dans ces derniers, la fonction d'abstraction est simplement une transposition d'un système de représentation orienté machine vers un système de représentation mieux adapté à la manipulation cognitive de l'utilisateur.

Cette simple abstraction nécessite cependant de réaliser un ensemble de traitements et de vérifications dont le but est d'assurer la conversion correcte du système de représentation. Ainsi, dès l'origine, les processus de traitement de langages peuvent être vus comme une opération de transformation de niveaux d'abstraction différents dont la propriété est de conserver la sémantique du traitement, c'est à dire sa signification exacte en termes de successions d'opérations réalisées par le processseur physique. L'introduction d'abstractions a eu donc historiquement pour but de faciliter le raisonnement du programmeur, en lui permettant d' élaborer un modèle personnel du fonctionnement de son programme. La totalité de l'évolution qui suivra peut se ramener à une dialectique fondamentale : d'une part la complexification croissante des architectures physiques (processeurs, co-processeurs, parallélisme, distribution,...), et logicielles (programmes de plus en plus vastes, modélisations de plus en plus riches) à laquelle on oppose des abstractions croissantes, visant à simplifier les tâches de spécification. Tous les programmeurs ont un modèle de ce que représente une instruction telle que i :=i+1, cependant aucun n'est capable de se représenter la liste exacte des instructions qui seront réalisées par le processeur, ainsi que les adresses mémoire concernées lors de l'exécution. Cet exemple illustre également un autre interêt des abstractions introduites par les langages : la possibilité de se détacher des contraintes induites par les jeux d'instructions du niveau zéro. Ainsi, porter un langage d'une architecture vers une autre revient à établir une

[1] nous entendons ici par processeur, tout dispositif physique au sens large, capable d'interpréter un jeu d'instructions par modification déterministe de ses états internes

traduction qui préserve la sémantique de l'ensemble des constructions composant le langage. Toutefois, il convient dès à présent de souligner les inconvénients potentiels des abstractions : en dissimulant la complexité sous-jacente des traitements, elles peuvent induire des erreurs cachées et des incertitudes, ou encore restreindre considérablement l'expressivité, c'est à dire la possibilité d'exprimer un grand nombre de traitements à partir de combinaisons d'un ensemble fini d'instructions.

Avec les langages dits *impératifs* tels que *COBOL, FORTRAN, PASCAL,* introduits dans les années 1950, les fonctions d'abstractions deviennent bien plus complexes et structurées, touchant à la fois les structures de données et les structures d'exécutions. Ces langages proposent à l'utilisateur la notion de types de données auxquels sont associés des opérateurs. Ainsi les langages tentent de rendre explicites des contraintes présentes naturellement dans les domaines applicatifs : un programme de comptatibilité produirait une erreur flagrante en additionnant des francs avec des centimes. Avec l'introduction des types, un progrès considérable se réalise : on isole une classe importante d'erreurs potentielles que le langage tente de détecter au mieux lors de la compilation du programme, sinon lors de son exécution. Au niveau des structures d'exécution, les notions de *procédures* et *fonctions* sont introduites, permettant d'encapsuler des traitements et d'isoler les fonctionnalités des programmes. Outre ce gain de clarté au niveau du modèle, ces notions introduisent la possibilité remarquable de simplifier les programmes en réutilisant au mieux des *sous-programmes*. La factorisation du code source offre ainsi la possibilité d'améliorer la couverture des tests et de simplifier la mise au point. L'étape suivante dans l'évolution des langages se caractérise par des hypothèses plus fortes sur les types de données et l'introduction de comportements algorithmiques dans les abstractions d'exécution. Ainsi, *LISP* [110] repose fortement sur les arbres binaires, le traitement fonctionnel récursif et des algorithmes de gestion de mémoire spécialisés dans les B-arbres.

Avec l'arrivé des langages *fonctionnels,* c'est une branche nouvelle qui se développe sous l'impulsion de la théorie du *lambda-calcul*, fécondant largement l'ensemble des disciplines associées aux langages. Le lambda-calcul, de part la pureté de sa syntaxe et de sa sémantique, se révèle être un support idéal des études théoriques menées sur les systèmes de types [24, 85, 43]. Dans ces langages, les abstractions d'exécutions identifient les constructions itératives à des appels récursifs de fonctions, comparables aux définitions mathématiques utilisant la récursivité. Le principal avantage réside dans la simplication amenée par la suppression des constructions itératives des langages impératifs. Ils demandent en revanche d'adopter une vision "mathématique" dans la résolution des problèmes de programmation. A ce prix, un langage fonctionnel élimine les effets de bord liés à la notion de variables globales inhérentes aux approches de plus bas niveau [52]. De plus, la simplicité du modèle d'exécution autorise des traitements statiques sophistiqués et des contrôles de types évolués et performants. Il est significatif que l'inférence de types ainsi que les modèles de traitements des exceptions aient été théoriquement étudiés puis implantés dans le langage *ML* [84].

Le langage *SIMULA-67* introduit la notion d'objets regroupant conjointement des données et des *méthodes* susceptibles d'agir sur ces données. L'objet se révèle être à la fois un moyen d'unifier les concepts de données et de traitements, tout en offrant un modèle très adapté à la représentation des connaissances pratiques. De fait, avec la naissance du modèle objet, c'est une nouvelle branche qui se développe avec succès, intégrant nombre d'aspects variés dans un même creuset : La théorie des types, et plus particulièrement les relations de sous-typage et de polymorphisme, une méthodologie de conception basée sur une catégorisation hiérarchique assez naturelle, et aussi des aspects génie logiciel liés à la réutilisation des classes au moyen de l'héritage et de la surcharge des méthodes. Ainsi, il est légitime de parler du "paradigme" objet, et de le considérer non pas comme une révolution, mais plutôt comme une évolution significative, ayant essaimé dans d'autres branches des langages de programmation, comme *Common Lisp Object System (CLOS)* qui incorpore la notion de multi-méthodes, ainsi que, très proba-

blement, la prochaine version de *ML-2000* ([21]) prendra en compte certains des concepts objets. Le langage *SMALLTALK-80* [51] propose une approche totalement dynamique du système de types et surtout un couplage fort avec l'environnement. Plus tard *EIFFEL* [111] propose un typage statique allégé par une technique d'inférence de types et une sémantique formellement définie. Notons que des travaux récents proposent une extension parallèle du langage sur des bases similaires [10].

Le langage *PROLOG* [34], représente un extrême dans l'abstraction d'exécution : se reposant sur un algorithme d'unification et une technique de résolution particulièrement efficace [101, 72], il vise à offrir au programmeur une vision purement logique de ses traitements. Si cette approche révèle son efficacité dans les domaines applicatifs pouvant être aisément décrits par des clauses logiques, elle montre également ses insuffisances dès lors que les traitements exigent de contrôler plus étroitement les étapes d'exécution ou de prévoir plus finement les performances temporelles des programmes. Les langages de contraintes [33, 28, 32] représentent une autre approche basée sur des abstractions d'exécution très forte, les résolveurs de contraintes, qui sont capables de montrer une grande efficacité dans des domaines applicatifs bien identifiés, mais qui restent encore, à l'heure actuelle, mal couplés aux langages dits "généralistes".

Avec ces extrêmes, la recherche d'un langage idéal qui semblait être un objectif implicite, paraît désormais se transformer en une approche plus pragmatique dans laquelle les qualités propres du langage ne sont plus considérées comme centrales pour résoudre les différents problèmes relatifs à la production de logiciels. Si ces qualités restent déterminantes, des facteurs conjoints tels que les outils de l'environnement, l'adéquation du modèle proposé par le langage aux domaines applicatifs, deviennent tout aussi déterminants. De plus, d'autres facteurs plus périphériques tels que l'extension des réseaux informatiques, des besoins en répartition d'applications et la nécessité économique d'intégrer des logiciels hétérogènes, font évoluer la vision des propriétés attendues des langages susceptibles de satisfaire ces exigences émergentes. En conséquence, de nouvelles abstractions apparaissent en réponse à ces besoins nouveaux. Cependant, il est remarquable qu'elles prennent forme dans des langages "intermédiaires" qui se proposent de résoudre uniquement des sous-problèmes et de faciliter la coopération entre d'autres langages dont le rôle reste plus classique. Ainsi, le langage de description d'interface de *CORBA* [55] est destiné à faciliter la distribution d'applications écrites dans des langages hétérogènes et propose une expressivité réduite à ces seules fins. On peut citer également la famille des langages d'interconnexion de modules ou de description d'architectures [108, 77], et bien d'autres encore. *OLAN* [14] est un des représentants les plus récents de cette famille de "sur-langages". *CLF* [6], dédié à l'intégration d'objets distribués via des schémas de coordination explicites en est un autre exemple.

Il est intéressant de citer deux langages récents conçus pour répondre à un certain nombre de problèmes identiques en produisant des solutions présentant sur certains points de très fortes différences. *JAVA* [80] tout comme *PYTHON* [102] reposent sur le concept de machines virtuelles afin de favoriser la portabilité des applications. Les machines virtuelles sont des programmes permettant d'émuler un processeur universel sur n'importe quel processeur physique. C'est un concept relativement ancien, introduit par exemple avec les premiers compilateurs *PASCAL* ou également avec *SMALLTALK*. Ainsi, le transport de la machine virtuelle sur différentes architectures physiques permet d'assurer directement le transport des applications. Toutefois, pour *JAVA* et *PYTHON*, un effort particulier est réalisé afin d'offrir une abstraction du système d'exécution et des librairies associées qui puissent présenter les mêmes propriétés. La machine virtuelle *JAVA* présente cependant une particularité relative à la sécurité : le code peut être vérifié au moment du chargement pour garantir un certain niveau de qualité et de sûreté à l'exécution. Aussi, cette propriété permet d'envisager la distribution et la mobilité du code sur des bases consolidées. La machine *PYTHON* quant à elle, offre un mode d'exécution restreint dans lequel le code non certifié comme sûr ne peut pas réaliser les opérations jugées sensibles sur la machine hôte. Si les deux approches

présentent des similarités assez marquées au niveau inférieur, les abstractions introduites au niveau langage divergent fortement. *JAVA* illustre le pragmatisme évoqué précédemment : le langage se présente comme un *C++* dans lequel toute irrégularité syntaxique ou sémantique aurait été élaguée. Toutefois, quelques concepts clés comme le glanage automatique de la mémoire, l'absence de préprocesseur, un modèle de processus légers bien integré, l'aide à l'annotation de commentaires et une plus forte structuration des modules sources ont été introduits. La seule originalité du modèle objet proposé est de faire coexister la notion *d'implémentation d'interfaces* et *d'extension de classes* de manière à contourner les complications entraînées par l'héritage multiple. L'ensemble se présente comme un grand nombre de librairies organisées autour d'un noyau langage peu original mais réutilisant au mieux les connaissances de langages antérieurs tels que *C* et *C++* .

PYTHON se présente comme un langage de prototypage, c'est à dire favorisant au mieux la rapidité des spécifications. Basé sur un système de types totalement dynamique, la phase de compilation est ramenée à une analyse lexicale et syntaxique très rapide, suivie d'une génération de *byte code* sans vérification et sans optimisation, tout aussi rapide. Les abstractions d'exécution ne présentent aucune originalité particulière, mais s'avèrent être une synthèse remarquable de diverses tentatives expérimentées par ailleurs. Elles semblent appliquer avec succès la théorie du rasoir d'Occam, tranchant toujours en faveur de la simplicité maximale. Les constructions syntaxiques sont particulièrement bien intégrées à un modèle de donnée central et fortement polymorphe, constitué de chaînes de caractères, de listes, de tuples et de dictionnaires (tableaux associatifs basés sur des fonctions de hachage performantes). De plus, ce noyau d'abstraction de structures de données est rendu extrêmement efficace au moyen de fonctions de conversion intégrées : un dictionnaire peut construire une *liste* de ses clefs d'entrée, une *liste* des valeurs contenues, ou une *liste* des *tuples* clés/valeur le définissant. Un modèle d'exceptions présentant les mêmes qualités de simplicité et d'intégration aux données permet de contrôler les cas d'erreurs à l'exécution. *PYTHON* intègre un modèle à objets de manière souple, pouvant coexister avec une approche plus procédurale ou fonctionnelle.

Ainsi, deux langages récents, à vocation généraliste, se situent dans la continuité de l'évolution, tout en se montrant en rupture avec un certain nombre de principes. *JAVA* sacrifie une certaine expressivité et souplesse à la rigueur d'un système de types contraignant, *PYTHON* explore la voie de la souplesse et la liberté maximales en s'appuyant sur la qualité de ses abstractions. Ces deux approches nous rappellent que de nombreuses voies restent à explorer dans la conception des langages en marge des progrès déjà réalisés. Toutefois, il est généralement admis que la qualité des environnements associés deviendra prépondérante dans l'appréciation des performances liées à un langage. Le regain d'intérêt constaté chez les industriels outre-Atlantique pour le langage *SMALLTALK* est une vivante illustration de cette tendance.

2.3 Les environnements de développement

Les environnements de développement comprennent tout outil conçu pour simplifier la production de programmes et par là-même, augmenter la productivité du programmeur. On peut catégoriser ces outils selon qu'ils servent à assister les différentes phases du développement ou à faciliter la gestion globale d'un système de logiciels [98]. On place traditionnellement dans la première catégorie les *éditeurs de programmes*, les *compilateurs*, les *éditeurs de liens* et *chargeurs*, les *préprocesseurs*, les *analyseurs de références croisées*, les *metteurs au point de haut niveau* et enfin les *aides au réglage* et à la *mise au point dynamique* (surveillance des allocations mémoire, profileurs d'exécution). On place dans la deuxième catégorie les outils de génie logiciel tels que les *configurateurs de systèmes*, *les gestionnaires*

de versions, les *éditeurs d'architectures*, les *générateurs de code source* et enfin les *générateurs de tests* et les *vérificateurs statiques*.

Il est évident que leur évolution a suivi étroitement celle des langages de programmation et des systèmes d'exploitation. Dans la première catégorie, des étapes significatives sont associées à *Smalltalk* et son environnement fortement intégré aux outils et concepts objets (éditeur graphique, metteur au point de très haut niveau, inspecteur de classes) puis à *Pecan* [96], démontrant l'intérêt des techniques de visualisation associées à la création de logiciels. On pourrait citer également *Interlisp* [113] conçu pour favoriser la programmation exploratoire en *Lisp* et utilisant intensivement la représentation arborescente des abstractions liées aux structures de données.

Avec la complexification croissante des systèmes logiciels, principalement en termes de volume des sources associées à des tâches de plus en plus vastes, on évolue d'une programmation détaillée ("programming-in-the-small") vers une programmation globale ("programming-in-the-large") et même plus récemment vers une programmation coopérative ("programming-in-the-many"). Dès lors, surgissent les problèmes d'intégration liés à la diversité des outils impliqués et à la nécessité de partager de nombreuses connaissances impliquées dans les diverses phases du traitement du logiciel. On distingue trois principaux axes d'intégration de niveaux d'abstraction croissants : intégration au niveau de la présentation (les fonctionnalités des outils sont accédées au travers d'une interface homme-machine commune), intégration au niveau des données (les outils partagent et échangent des structures de données utilisant un format commun) et intégration au niveau du contrôle (les outils échangent des évènements et des informations d'exécution au travers de mécanismes communs).

Les environnements actuels développent de manière plus ou moins marquée ces trois axes et rencontrent de nombreuses difficultés dues au surcoût de développement entraîné par l'inadéquation des outils aux problèmes d'intégration [74]. En effet, les compilateurs sont au centre des processus de développement, et ils ne sont pas conçus de manière "ouverte", c'est à dire de manière à offrir des facilités de réutilisation ou de mise en oeuvre partielle dans des contextes multiples. Il n'est généralement pas possible d'utiliser la partie frontale d'un compilateur afin de générer une représentation syntaxique abstraite susceptible de constituer l'entrée d'un outil tel qu'un formateur de texte source, extracteur de commentaires ou un analyseur-vérificateur statique. Pour un tel outil, il serait souhaitable de disposer de représentations internes plus "profondes", telles que un graphe décoré du flot de données. Outre l'aspect fonctionnel, fortement lié à la définition de structures de données pivots possédant différents niveaux de richesse sémantique, la coordination des noyaux de traitement nécessite un travail d'emballage ("wrapping") d'autant plus important que les fonctions de traitement du langage sont pauvrement isolées. Si la définition d'architectures ouvertes et réutilisables est aujourd'hui un problème générique, il existe toutefois des raisons historiques justifiant la conception monolithique des compilateurs. La rapidité de la phase de compilation a pendant longtemps été considérée comme le critère qualitatif primordial conditionnant l'ensemble des techniques mises en oeuvre. Ainsi, dans le contexte d'un développeur unique utilisant une station de travail sans commune mesure avec les puissances de traitement actuelles, il est impensable de pénaliser lourdement la compilation par la génération de multiples formats intermédiaires. De plus, la manipulation de ces structures, riches par définition, entraîne un surcoût considérable dans le développement du compilateur qui, rappelons le, pour des raisons d'efficacité, est écrit généralement dans des langages d'assez bas niveaux tels que le *C*.

Nous arrivons ici à un point fondamental : la conception du cycle de développement. Il est essentiel d'observer que les techniques de traitement du langage reposent sur un cycle à trois phases : édition, compilation, détection des erreurs. Chaque phase étant exclusive, il est nécessaire de les enchaîner linéairement et donc d'additionner chaque temps de traitement pour converger vers un résultat jugé satisfaisant. Dans cette perspective, il devient primordial de minimiser les phases qui ne dépendent

pas de l'utilisateur. Or, en imaginant une coopération plus étroite entre l'édition et la vérification syntaxique, par exemple, nombres d'erreurs superficielles peuvent être corrigées au plus tôt, économisant ainsi de nombreux cycles.

Ce type d'approche a été largement expérimenté dans les *éditeurs syntaxiques* tels que le *SYNTHETIZER GENERATOR* [99], et appliqué dans le générateur d'environnement *CENTAUR* [19]. Si la voie ouverte par les éditeurs syntaxiques nous a montré que les aspects liés à l'interaction avec l'utilisateur et à l'ergonomie en général sont déterminants pour l'efficacité de ces outils, il reste cependant à établir l'étape ultérieure dans l'utilisation des connaissances "profondes" du compilateur. Comment utiliser des informations fortement contextuelles pendant le processus d'édition ? Comment, par exemple, utiliser des règles de portée dans les définitions de variables ? Comment amener dans l'espace de travail de l'utilisateur des informations pertinentes, c'est à dire correctement reliées à la sémantique statique du langage considéré ? A ces aspects "individuels" du processus de production de logiciel, il convient de prendre en compte les aspects "collectifs" impliqués dans les technologies de génie logiciel. Comment favoriser l'intégration des noyaux de traitement (vérifications syntaxiques, contrôle des types, vérification sémantique, optimisation) dans des systèmes coopératifs, extensibles et modulaires ? Comment obtenir un couplage fort de la connaissance dans un système faiblement couplé ? Comment permettre des évolutions des outils de traitement sans introduire de problèmes de compatibilité ascendante ? Sans vouloir répondre systématiquement à l'ensemble de ces questions liées à de nombreux domaines de recherche, deux axes d'études doivent être combinés afin de progresser dans ces directions. D'une part, générer les noyaux de traitement à partir de descriptions de haut niveau, d'autre part concevoir systèmatiquement les outils comme des compositions de noyaux modulaires. Ces deux aspects seront conjointement pris en compte dans l'étude du langage *CIRCUS*, objet de ce travail de recherche. Le chapitre suivant proposera une étude approfondie de l'approche menée dans le projet *CENTAUR* qui explore la génération d'environnements intégrés. Les problèmes de composition, quant à eux, peuvent être éclairés par les travaux généraux portant sur la conception architecturale des logiciels vus comme des hiérarchies de composants en interaction [77, 14, 68], ou encore, d'un point de vue plus dynamique, par les travaux récents sur les langages de coordination [88, 29].

D'autre part, si l'interface homme-machine semble jouer un rôle de plus en plus important, notamment au niveau de l'intégration, il est légitime de se demander si de nouveaux langages pourraient se réveler plus adaptés aux possibilités nouvelles offertes par les performances graphiques et les facilités intéractives des ordinateurs actuels et futurs. La section suivante propose une analyse des travaux portant sur les langages visuels, afin de déterminer les points faibles et forts de ces approches, et surtout, de définir leur potentiel.

2.4 Les langages visuels interactifs

Définition.
Un langage visuel possède les mêmes caractéristiques qu'un langage textuel c'est à dire qu'on peut le décrire en termes d'unités lexicales, de syntaxe et de sémantique. La différence réside dans le système de représentation permettant de coder l'alphabet de base. Ainsi, si le seul moyen de construire des mots est de juxtaposer des lettres de l'alphabet, on peut visuellement élaborer des formes en utilisant de nombreuses dimensions telles que la couleur, la forme, la position.

Par la suite, il est naturellement possible d'associer aux constructions syntaxiques visuelles une sémantique, c'est à dire une signification précise relative à un système capable d'interpréter le code. La difficulté pour le concepteur d'un langage visuel est de connaître les particularités des différentes

variables visuelles constituant le système de représentation, afin de conférer une perception intuitive de la sémantique sous-jacente.

On peut distinguer sept variables visuelles de base [16] : la position, la taille, la forme, l'orientation, la couleur, la texture, l'intensité. Ces paramètres sont perçus simultanément par l'oeil mais traités différemment par le cerveau en fonction de leur nature. Il est possible d'ordonner les informations de taille, d'intensité et de position, de comparer les informations de couleurs, d'orientation, de texture et de formes. Concrètement, si par exemple on transpose une information telle que la taille mémoire occupée par un objet dans la variable de forme, il sera impossible à l'utilisateur de percevoir si le carré consomme plus de ressources que le triangle. Si le système de signes se révèle potentiellement beaucoup plus riche qu'un alphabet littéral, il se montre donc également plus délicat à utiliser. Il convient toutefois de remarquer que, dans un langage textuel, seul un certain nombre de mots *significatifs* sont construits à partir d'un alphabet donné, et que des propriétés phonétiques similaires aux propriétés visuelles ont été instinctivement prises en compte lors de son élaboration. Dans les travaux les plus récents, comme le générateur de langages iconiques *Vampire* [76], il est possible de définir la sémantique au niveau visuel lui-même. Les icônes sont spécifiées indépendamment à l'aide d'un éditeur spécialisé. Elles constituent un alphabet de base dont les signes peuvent être complexes et incorporer des attributs textuels de types variés. Des règles de réécriture permettent de définir l'évolution dynamique des constructions visuelles, et s'apparentent aux sémantiques opérationnelles de type "petit pas" (*small-step semantics*). Le membre gauche permet de préciser les paramètres devant être unifiés simultanément, et le membre droit la façon de les transformer. Les figures 2.1 et 2.2 montrent une spécification assez complexe, et la figure 2.3 montre un programme exécutable selon cette sémantique. Chaque membre possède deux fenêtres de définition, une pour les attributs et le code d'évaluation *Smalltalk* associé, l'autre pour les paramètres purement visuels. De plus, une priorité d'évaluation est assignée à chaque règle afin d'ordonner les productions multiples. Le premier groupe de règles de la figure 2.1 décrit l'écoulement du flot de contrôle, symbolisé par un carré en surimpression entourant l' icône "active". Les deux premières productions orientent le flot du haut vers le bas, et de la gauche vers la droite. Les deux suivantes établissent les conditions de lancement et d'arrêt du flot de contrôle, chacune sur une icône dédiée.

Le deuxième groupe de règles (figure 2.2) précise comment saisir textuellement une valeur d'entrée, comment afficher une variable de sortie, et comment réaliser un calcul numérique simple. Ces règles s'appliquent sur les icônes "actives" possèdant le type visuel requis. La figure 2.3 montre un programme valide pour cette spécification sémantique. Le lecteur appréciera le potentiel expressif d'une telle approche, et imaginera peut-être des programmes déroulants plusieurs flots d'exécution tout en conservant une bonne intelligibilité. Bien entendu, ce type d'approche ne permet pas de résoudre la même classe de problèmes qu'un langage textuel compilé, mais ouvre de nouvelles perspectives sur des traitements purement visuels de haut niveau. Il est à noter que dans *Vampire*, la syntaxe visuelle est confondue dans le traitement sémantique. Plus précisément, aucun contrôle syntaxique n'est réalisé, ni avant l'exécution, ni pendant. Un programme syntaxiquement incorrect (icônes non juxtaposées par exemple) verra son exécution interrompue par défaut de production applicable. C'est donc une approche purement interprétative qui est décrite ici. Une approche compilée devrait imposer de décrire une syntaxe séparément. Elle compenserait la perte de souplesse par des possibilités de générer du code machine de bas-niveau pouvant être beaucoup plus efficace.

Évolution historique : un domaine jeune en expansion.

Les relations entre les langages visuels et textuels sont historiquement complexes. On s'attache souvent à leurs différences afin de mieux les définir et les contraster. Pourtant, jusqu'aux développements récents des grammaires visuelles, leurs frontières n'étaient pas formellement établies. Une tentative étonnante de formalisation visuelle est décrite en [22], par un chercheur devenu plus tard un contributeur important

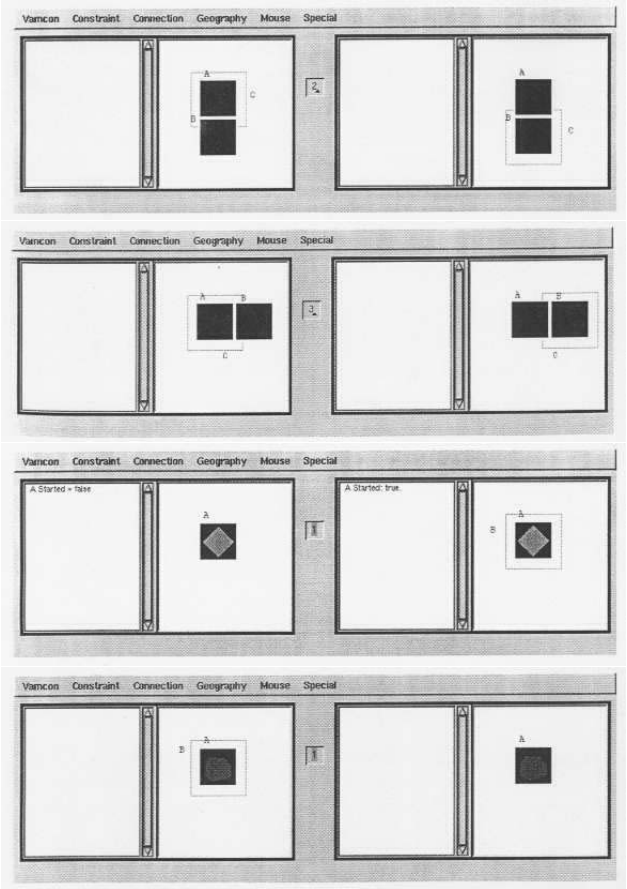


FIG. 2.1 : Une spécification sémantique en *Vampire* : les règles générales décrivant le flux de contrôle.

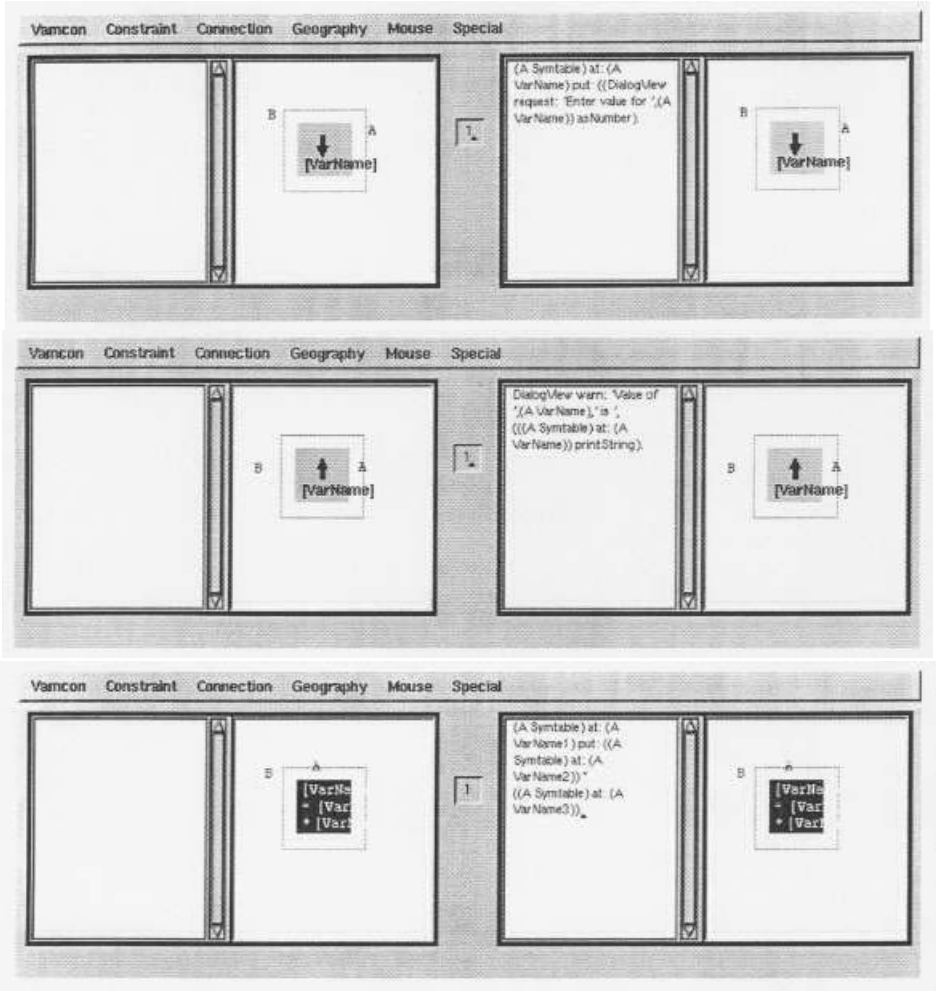


FIG. 2.2 : Une spécification sémantique en *Vampire* : les règles décrivant le traitement des attributs

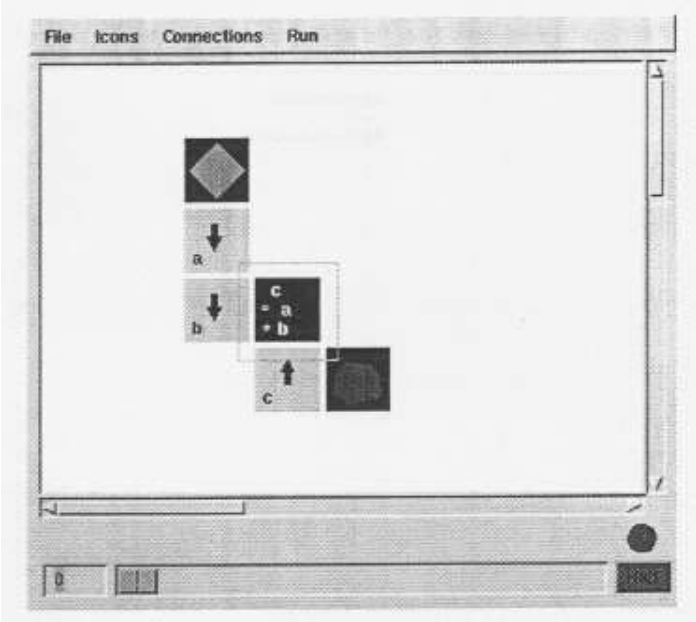


FIG. 2.3 : Un programme dont l'exécution est contrôlée par une spécification sémantique *Vampire* de haut niveau

de la théorie des types en informatique. Cette tentative est très représentative de cette difficulté à intégrer les différences fondamentales, liées davantage à la sémiotique qu'à la sémantique. Ainsi, Luca Cardelli propose une transposition graphique fort peu intuitive de la syntaxe d'un langage fonctionnel de type *ML*. Le résultat relève plus d'une notation mathématique enrichie spatialement que d'une approche exploitant les capacités cognitives humaines dans le domaine du visuel. Il est important de souligner que les premiers systèmes graphiques commençaient seulement à bénéficier de l'affichage haute résolution et de possibilités d'animations. Les précurseurs d'alors devinèrent le potentiel offert par les évolutions technologiques qui débutaient seulement. Les premières expérimentations des années 80 proposent en fait une *abstraction visuelle* des principales constructions des langages, comme les diagrammes de flots (organigrames) [12] ou C^2 [67]. Dans ce dernier, un sous-ensemble du langage C est transposé visuellement au moyen de la méthaphore du puzzle. Un nombre restreint de pièces représentent les blocs d'instructions (contenant d'autres constructions), les conditionnels *si-alors* et *si-alors-sinon*, les itérateurs *for* et *while*. Les contraintes morphologiques des pièces expriment les contraintes syntaxiques de *C*. C^2 est conçu comme un langage mixte textuel/visuel. De fait, deux vues différentes sont maintenues en cohérence par le système en cours d'édition. La première permet d'éditer les programmes à l'aide de commandes *vi* couplées à un interpréteur. La seconde permet d'avoir une vue globale, et de composer des blocs à partir d'une palette de base. Cette idée de transposer des contraintes syntaxiques et visuelles dans le système de représentation constitue un excellent exemple de la tendance, s'affirmant regulièrement par la suite, de tirer parti plus profondément des propriétés spécifiques des systèmes de signes visuels, au niveau des lexèmes, de la syntaxe puis de la sémantique. Par ailleurs, l'idée développée dans C^2 de combiner le textuel avec le visuel s'impose comme un moyen pragmatique de tirer meilleur parti des spécificités de chaque approche. Pourtant, cette avancée ne semble pas avoir eu la place qu'elle méritait dans les travaux ultérieurs, hormis l'utilisation de vues multiples ouvrant sur différents niveaux sémantiques, déjà explorées dans le système *PECAN* [97] notamment.

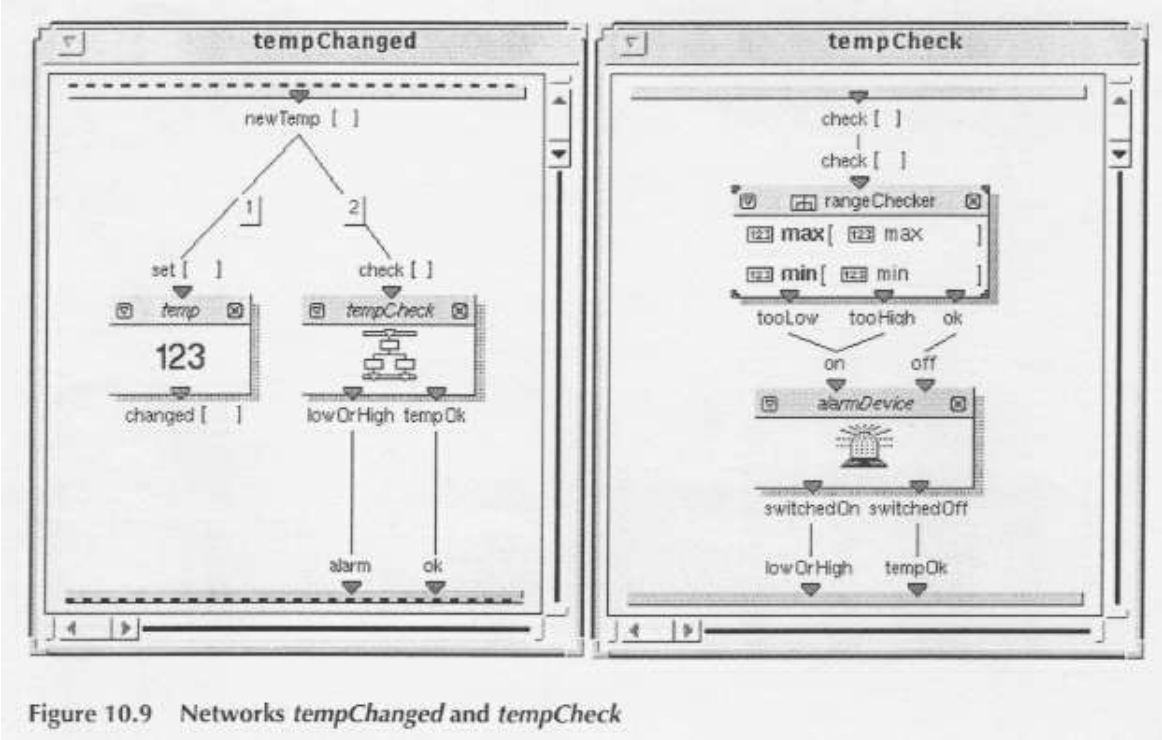


Figure 10.9 Networks *tempChanged* and *tempCheck*

FIG. 2.4 : programme visuel VISTA (type *flot de données*)

Par la suite, de nouveaux langages exploitent plus complètement les spécificités du système de représentation, au prix il est vrai d'abstractions d'exécution importantes entraînant des hypothèses d' exécution assez fortes. Ce sont, globalement, des langages basés sur les flots de données tels que le célèbre *PROGRAPH* [41] ou *FABRIK* [75], dont les éléments lexicaux de base sont interconnectés par des segments et forment des graphes de traitement de complexité croissante. La figure 2.4 montre un exemple représentatif de cette famille, où les traitements sont réalisés dans la continuité de l'écoulement des données. La figure 2.5 montre la spécification en *LABVIEW* d'un télémètre complexe, composé de sous-systèmes de mesure et de traitement physique étroitement associés à des instructions et procédures d'acquisition.

Avec le développement d'applications visuelles de tailles plus significatives, un problème majeur se profile alors dans l'étude des langages visuels : la résistance au facteur d'échelle ("*scalability*"). Les représentations ne se prêtent pas à la manipulation de larges spécifications. Ou plus précisément, ce type de contrainte semble s'opposer aux propriétés *extensives* des informations visuelles (en opposition aux propriétés "compréhensives", voire même symboliques, des langages textuels). Ainsi, lorsque les signes manipulés dans un programme visuel restent en dessous d'une limite quantitative, qui varie fortement en fonction des choix de base et des personnes, ils sont perçus dans leur ensemble (perception fortement *globale*) aussi bien que dans le détail de leurs relations.

Passé ce seuil, soit les signes deviennent imperceptibles dans leurs détails, soit la perception d'ensemble devient impossible. Avec la perte de cette caractéristique, c'est le point fort essentiel qui semble être remis en cause.

En fait, c'est une loi d'ordre général qui veut que les manipulations intellectuelles ne se réalisent

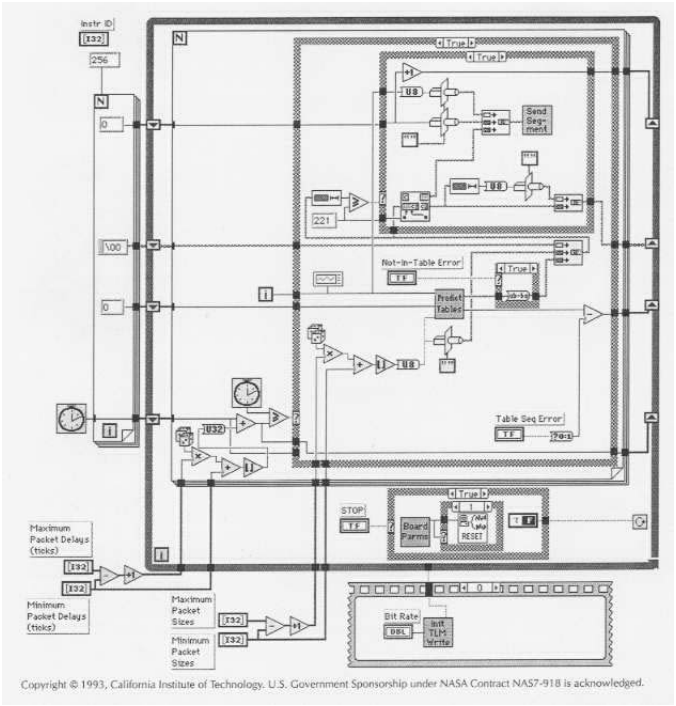


FIG. 2.5 : programme visuel LABVIEW

efficacement que sur un nombre restreint de concepts. Les travaux en psychologie cognitive ont établi que ce nombre est en relation étroite avec la mémoire à court terme, dont la profondeur temporelle varie entre quelques secondes jusqu'à une minute environ, inversement proportionnelle aux quantités d'informations mémorisées [40]. La mémoire à court terme est considérée comme la zone de travail de la pensée attentive. Cette dernière est sollicitée dès lors qu'une représentation complexe doit être conceptualisée afin d'élaborer des structures simplificatrices. Le même problème se pose dans l'approche textuelle, et même beaucoup plus bas dans l'échelle de la complexité. Les solutions éprouvées sont l'encapsulation hiérarchique des fonctionnalités à l'intérieur de modules, de procédures, de classes, dont la complexité interne est dissimulée et résumée par de simples noms. Dans les langages visuels, des méthodes de structuration adéquates doivent être proposées parallèlement. Certains auteurs proposent des solutions basées sur les propriétés géométriques des formes dont la taille peut varier dynamiquement en fonction de l'occupation de l'espace de représentation. L'information reste toutefois accessible au prix d'un minimum d'actions de l'utilisateur.

Ainsi dans *VIPR* [30] les signes de base sont constitués de cercles pouvant être superposés puis récursivement dilatés ou contractés de par les facilités géométriques offertes par ce type de constructions radiales. Le figure 2.6 montre une representation *VIPR* d'un programme utilisant des constructions complexes comportant des références sous forme de flèches. Il est possible d'observer les variations de dimensionement des différents groupes graphiques, en fonction des contraintes de placement. Nous reviendrons plus loin sur les particularités dynamiques de ce langage.

Généralement, les emboîtements graphiques sont utilisés en tant que transposition directe de l'encapsulation logique de sous-structures. De plus l'espace de travail est logiquement sans limite, et des fonctions de zoom permettent d'explorer les détails devenus imperceptibles dans la vue principale. D'autres

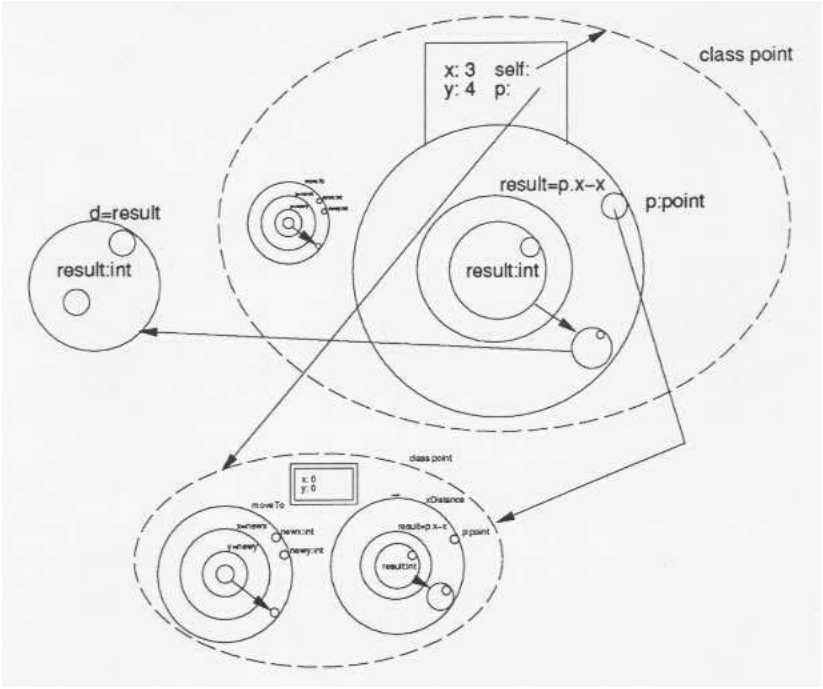


FIG. 2.6 : Un programme *VIPR*.

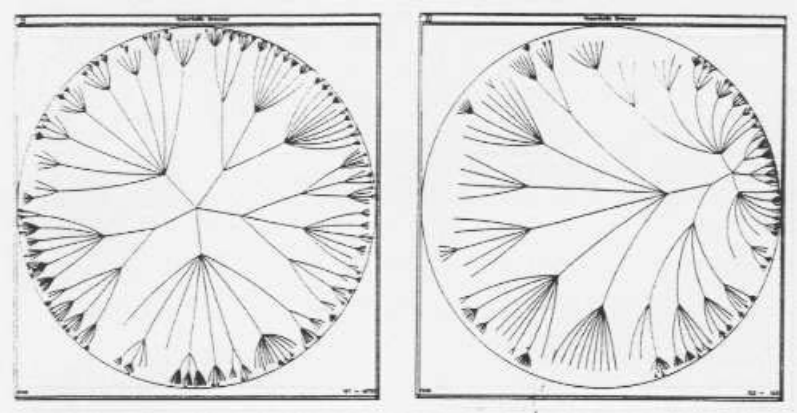


FIG. 2.7 : Arbre hyperbolique contenant 1004 noeuds , d'abord centré, puis continuement déformé pour prendre en compte un nouveau centre d'intérêt

auteurs [103] proposent des techniques de loupes déformantes (*fisheye views*) qui permettent de grossir les zones d'intérêt sans perdre la perception du contexte d'ensemble. Dans *hyperbolic browser* [69], des arbres complexes sont visualisés en exploitant des déformations spatiales contrôlées qui dilatent le centre d'intêret (voir fig. 2.7). Les mouvements de la souris permettent de déplacer ce centre d'intêret en préservant la continuité de la perception. Ce type de *Browser* est aujourd'hui commercialisé par la société *InXight* sous forme de composants, adaptables notamment à la visualisation d'informations issues du *World Wide Web*.

De l'expressivité.
L'expressivité d'un langage correspond intuitivement à sa capacité à modéliser un grand nombre de configurations sémantiques. Il serait plus précis de considérer un rapport entre la quantité de sémantique codée dans le langage et la quantité de signes et d'informations syntaxiques utilisées pour réaliser cet encodage. Et encore, ce critère semble d'avantage refléter la *concision* du langage. Il serait intéressant de considérer en outre la compréhension du système de signes, qui fait largement appel aux capacités personnelles du programmeur, bien que des règles générales de construction puissent certainement influencer des communautés très larges d'utilisateurs. Il ne semble pas exister de définition formelle du critère d'expressivité [1]. Pourtant, c'est un facteur primordial pour apprécier les qualités d'un langage. En considérant cette définition intuitive, il semble que les langages visuels possèdent un avantage initial décisif, lié à la simultanéité dans la perception des paramètres visuels. Une bonne représentation graphique permet de saisir des milliers d'informations et leurs corrélations d'un seul regard. Certains travaux ont exploré les propriétés expressives des systèmes de représentation tridimentionnels [119, 100] et démontré qu'ils étaient bien adaptés à la manipulation de grandes quantités d'informations. Ces systèmes exigent toutefois une prise en compte plus fine des contraintes temporelles nécessaires aux animations requises pour explorer les trois dimensions. Ces dernières doivent effectivement être projetées sur les deux dimensions des écrans standards, et l'opération inverse doit être réalisée pour les périphériques de pointage. Ainsi, un niveau d'abstraction spatiale supplémentaire se fait au prix d'un accroissement important de la complexité des traitements.

De l'interactivité et de la dimension temporelle.
Nous abordons ici un point de vue plus personnel, selon lequel il est intéressant de dissocier plus fortement les aspects interactifs, liés au temps, des aspects graphiques, liés à la perception visuelle. Les deux sont traités simultanément depuis l'origine des recherches dans ce domaine, et il est clair que leur combinaison conduit potentiellement à des résultats de très haute qualité. Cependant, une définition visuelle peut être totalement spécifiée par un processus d'édition quelconque, puis séparément analysée puis compilée. Dans ce cas, la structure temporelle liée au processus d'édition ne rentre en compte ni dans la syntaxe, ni dans la sémantique du langage, et le cycle de développement se ramène aux trois phases édition-compilation-tests evoquées précédemment. La dynamique de l'édition, elle, ne peut pas se décrire indépendamment du système de représentation. De plus, les actions de l'utilisateur se déterminent également en fonction des propriétés temporelles du système de signes : un bouton ne peut être activé qu'à l'intérieur de la fenêtre temporelle où il est perceptible à l'écran. De nombreux travaux abordent les représentations tridimensionnelles comme un moyen d'étendre l'expressivité des langages visuels, en augmentant potentiellement la densité d'information disposée dans l'espace de travail. De plus, une dimension spatiale supplémentaire augmente les possibilités de combinaison, et donc la richesse de la

[1] Il n'est pas question ici de l'expressivité relative à la calculabilité des langages, qui est un critère plus qualitatif que quantitatif

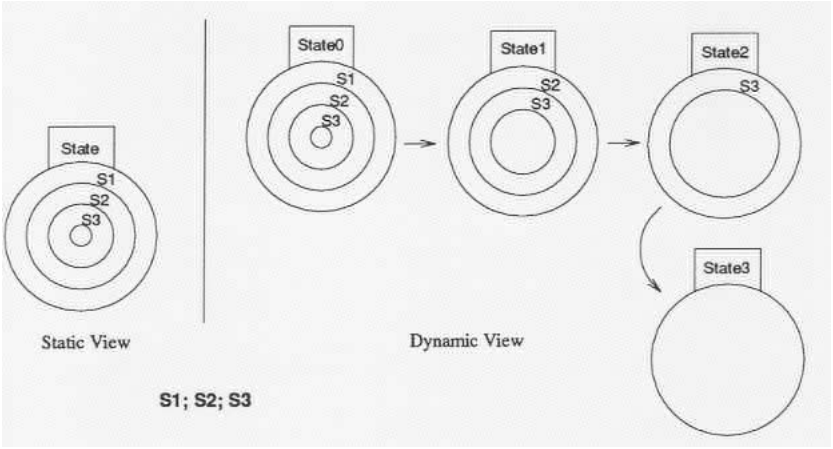


FIG. 2.8 : Une construction sequentielle en *VIPR*, et son animation à l'exécution

syntaxe spatiale. On peut citer [119], dans lequel un critère de *compacité* est proposé afin de comparer les systèmes de représentations 3D et 2D. Robertson, Card et Mackinlay [100] présentent *l'Information Visualizer*, intégrant de nombreux concepts novateurs dans une architecture originale appellée *coprocesseur cognitif*. Si les trois dimensions sont spectaculairement mises en valeur dans des représentations telles que le *cone tree* ou le *perspective Wall*, l'apport le plus essentiel semble relié à la prise en compte du temps psychologique de l'utilisateur. Dans la continuité des travaux antérieurs de Card sur les rapports de la psychologie cognitive et des interfaces homme-machine, trois temps qualitativement très différents sont identifiés et pris en compte séparément dans la conception de l'architecture. Le temps dit *perceptuel* correspond aux évènements dont la durée est inférieure à 100 millisecondes. Dans cet espace temporel, le cerveau résoud des problèmes de perception avec une "bande passante" très importante. C'est dans cet espace que des corrélations rapides sont établies et maintenues sur les paramètres visuels de bas niveaux. Le temps dit *interactif*, entre 100 millisecondes et 10 secondes, est consacré aux tâches de planification et à l'articulation de commandes plus ou moins complexes permettant de réaliser ces tâches. Le temps dit *à long terme*, supérieur à dix secondes, est utilisé pour faire réaliser des recherches étendues à un sous système qui travaille en parallèle de l'utilisateur et le décharge des attentes inutiles. Cette architecture a pour but de déplacer le plus grand nombre possible de charges cognitives dans les espaces perceptuels et à long terme. Ainsi, l'espace interactif est libéré afin d'optimiser l'efficacité de l'utilisateur. Le principe de continuité perceptuel est un excellent exemple du traitement spécifique des trois temps psychologiques. Les auteurs montrent que les rotations dans l'espace doivent être progressives afin de ne pas perdre les repères spatiaux de l'utilisateur. Un brusque changement, au contraire, perturbe l'observateur qui doit mobiliser son attention consciente pour interpréter le changement et restructurer mentalement son espace de travail.

Il apparaît clairement que l'espace temporel doit aussi être structuré afin de tirer partie des propriétés de la perception humaine. Par ailleurs, cet espace est également une dimension de la représentation utilisée pour animer des processus complexes. De nombreux travaux proposent des abstractions dédiées à l'animation d'objets graphiques [20, 47, 109, 26] et les utilisent pour réduire la complexité temporelle des algorithmes ou des programmes. Le langage *VIPR*, déjà abordé précédemment, utilise une transposition assez systématique des constructions des langages textuels, mais associé étroitement la vision statique du programme avec une vision dynamique de son exécution. La figure 2.8 illustre ces deux aspects : à gauche, une spécification statique représentant trois actions séquentielles ; le diagramme de

droite montre l'animation obtenue lors de l'exécution du programme. Cette approche possède l'avantage de rapprocher le modèle que se fait le developpeur de son programme, du modèle qu'il élabore pour son exécution. Ce schisme est considéré à juste titre comme un des problèmes majeur de la mise au point. Certains langages textuels, tels que les langages à objets, ont de fait accentué cette distance, en proposant des abstractions telles que les hiérarchies de classes, qui disparaissent à l'exécution, ou prennent des formes profondément différentes.

Synthèse.

Si l'étude des formalismes visuels est encore très récente, elle nous enseigne d'abord, par ses succès et ses échecs, que ses qualités et défauts semblent être complémentaires de ceux des langages textuels. Ainsi, la perception globale, analogique et quantitative des uns s'oppose à la perception détaillée, analytique et qualitative des autres. A une vision dynamique, on peut opposer une vision purement statique.

Elle nous enseigne d'autre part que l'expressivité des langages visuels semble être plus universelle que celle des langages textuels. Les structures sémantiques "profondes", dont Chomsky pensait qu'elles étaient la clef du traitement automatique des langues naturelles, pourraient se révéler plus facilement transposables et manipulables dans des systèmes de signes visuels et dynamiques. L'évolution d'un tel système est compréhensible par des utilisateurs de différentes cultures et de différents niveaux de connaissances.

Elle nous apprend également que la dynamicité des représentations visuelles, telle qu'elle est permise par les capacités de traitement graphique actuelles et futures, représente sans aucun doute l'innovation potentielle majeure des langages visuels. Nous commençons à peine à entrevoir la richesse des systèmes de signes capables d'une grande plasticité temporelle et d'établir une intéraction étroite avec l'humain [73].

La génération des langages visuels peut nous apprendre beaucoup sur la nature même de ces derniers. Elle peut également, dès lors que les formalismes adaptés et suffisamment expressifs sont établis, permettre de réaliser automatiquement un certain nombre de vérifications permettant de détecter des "erreurs" dans la conception du langage. L'enjeu est donc de pouvoir rendre opérationnel et systématiquement applicable un grand nombre de connaissances qui aujourd'hui tiennent plus de "l'artisanat éclairé" que de la technologie. De plus, il est raisonnable de penser que la génération amènera une économie importante des efforts de développements liés aux applications visuelles, à la condition que l'expressivité des spécifications permette de couvrir une large palette d'applications. Les enjeux macro-économiques sont aujourd'hui énormes, les techniques d'assistance à la conception et l'écriture des applications visuelles sont dans leur enfance.

2.5 conclusion

> *The progress of science involves a constant interplay between diversification and unification. Diversification extends the boundaries of science to cover new and wider ranges of phenomena ; successful unification reveals that a range of experimentally validated theories are no more than particular cases of some more general principle. The cycle continues when the general principle suggests further specialisations for experimental investigation.*
>
> From C.A.R. Hoare's position statement
> ("http ://www.acm.org/surveys/1996/HoareUnifying/")

Il semble en effet qu'après avoir produit une grande diversification, les progrès de la recherche actuelle

requièrent un important travail d'unification.

Dans le domaine général des langages de programmation, l'évolution s'exprime comme une montée croissante dans les abstractions concernant les structures de données, les structures d'exécution et les modèles associés. De plus, des divergences significatives se sont affirmées entre des branches comme les langages fonctionnels, orientés objet, impératifs, logiques ou à base de contraintes. Il faut considérer également une complexification *qualitative* des architectures physiques (hétérogénéité, parallèlisme, réseaux) et *quantitative* (extension des ressources informatiques, internationalisation de l'internet et du world wide web). L'expérience montre aujourd'hui que ces tendances sont durables, et que de plus, le passé de l'informatique présente une forte rémanence : *FORTRAN* et *COBOL* restent des acteurs de premier plan dans l'économie et dans le savoir-faire global. Ainsi les technologies de *reverse engineering* sont-elles des points clefs de l'évolution industrielle, comme nous le rappelle l'étonnant épisode de l'année 2000 Ainsi le syndrôme de Babel prend ici la forme d'une multiplication des langages, chacun ayant des spécificités lui permettant de s'imposer et de perdurer dans des domaines différents.

Pour les environnements de développement, le syndrôme de Babel apparaît comme une difficulté croissante à faire communiquer des outils de plus en plus pointus, difficiles à écrire et surtout à intégrer. La connaissance mise en commun doit à la fois être de plus en plus élaborée, en rapport avec les performances des outils, et à la fois de plus en plus transformable, afin de pouvoir circuler plus souplement et s'adapter plus aisément aux évolutions et ajouts de nouveaux outils.

Les langages visuels s'annoncent universels et intuitifs, mais nécessitent des abstractions graphiques et temporelles encore plus distantes des possibilités *sui generis* des processeurs et de leurs périphériques. Si autrefois, une instruction *print* ou *read* suffisait à assurer le dialogue avec l'humain, il faut aujourd'hui des milliers d'instructions pour réaliser une interaction jugée acceptable. D'autre part, le temps psychologique humain semble être infiniment éloigné des temps de cycle des ordinateurs, et le développeur désirant intégrer des animations doit faire face à un nombre de difficultés considérables et redhibitoires.

Les processeurs ont certes évolué dans leurs performances générales, mais aucunement dans leurs performances qualitatives. Les tentatives historiques visant à réaliser des architectures dédiées à des langages ou des modèles d'exécution ont échoué, sauf dans les cas particuliers de super calculateurs parallèles tel que la *connexion machine* offrant d'épouvantables difficultés de programmation générale. Les ordinateurs analogiques, optiques, organiques ou biologiques, ne sont pas encore sortis des laboratoires, et leur programmabilité ne peut pas être évaluée aujourd'hui.

Il apparait *in fine* de plus en plus nécessaire d'établir des théories, de rechercher des techniques, de proposer des solutions visant à améliorer les transpositions entre les différents niveaux d'abstraction, les différentes sémantiques impliquées de part et d'autre dans les traitements langagiers. Inventer des "super-traducteurs" afin de faire face au syndrôme de Babel tel qu'il a été introduit dans ce chapitre : c'est le cadre général dans lequel s'inscrit ce travail sur *Circus*, tentant de prendre en compte un certain nombre des paramètres et facteurs jugés déterminants pour cette contribution. La vocation de ce métalangage sera donc de générer des composants pour (i) prendre en compte la diversité des langages de programmation, (ii) traiter les différents niveaux sémantiques dans les transformations qui leurs sont associées, (iii) construire des architectures pour leurs environnements de developpements et (iv) mettre en rapport le contrôle de l'exécution avec les exigences de langages visuels fortement dynamiques.

Théorie et pratique des outils de génération

3.1 Introduction

Dans le chapitre précédent, nous avons proposé une vision externe de l'évolution des langages et des environnements associés. Nous abordons à présent l'aspect interne, c'est à dire, la manière dont les langages sont étudiés , spécifiés et réalisés. En arrière plan des nombreux progrès réalisés, la théorie est partout, précédant toujours les réalisations novatrices, balisant les chemins dangereux, identifiant les limites des approches calculatoires, caractérisant les algorithmes, proposant de puissants outils méthodologiques. Nous présenterons dans un premier temps quelques éléments permettant de situer et d'introduire les traits essentiels de ce vaste domaine. Nous examinerons ensuite de manière plus détaillée les travaux se rapprochant le plus de *Circus* et nous attacherons à mettre en lumière leurs points forts et leurs limites.

3.2 La théorie : au cœur de l'évolution des langages

3.2.1 Introduction

Les trois aspects évoqués, l'*étude*, la *spécification* et la *réalisation* sont toujours étroitement liés. L'étude d'un problème amène à élaborer des modèles mathématiques. Ceux-ci identifient les éléments du problème à des entités précises, dont les propriétés sont exprimées dans le modèle. Ce point de départ permet ensuite d'étudier les inter-relations, à l'intérieur du système formalisé, et parfois d'obtenir des résultats inattendus, la plupart du temps en tentant de prouver des hypothèses fausses. Le travail mathématique a alors permis d'élaborer une représentation correcte du domaine qui trouve sa première application dans des méthodes de spécification, c'est à dire, un moyen de structurer toute une classe de problèmes dans un vocabulaire adapté, capable de lever les ambiguités ou incertitudes inhérentes. Ces spécifications, à leur tour, peuvent conduire plus ou moins directement, à des réalisations susceptibles de résoudre les problèmes d'origine.

Un exemple significatif concerne l'élaboration de *CCS* ("*Communicating Concurrent Systems*") [81], théorie des systèmes concurrents élaborée par *Robin Milner*, qui a obtenu le prestigieux *Alan Turing Award* pour l'ensemble de son apport scientifique. Robin Milner se propose, en 1972, d'étudier les problèmes liés à la sémantique des langages parallèles, en utilisant les modèles *fonctionnels*, appliqués alors aux programmes séquentiels, conçus comme des applications successives de fonctions sur les états mémoire. En découvrant que le parallélisme induit un indéterminisme fondamental, non réductible au modèle fonctionnel de la mémoire, Milner développe une approche théorique *observationnelle* au moyen d'un calcul faisant apparaître les propriétés structurelles des systèmes concurrents. L'étude de ces propriétés débouche sur une caractérisation fine et riche de la notion d'équivalence comportementale, c'est à dire, les bases qui permettent de comparer les entités décrites [1]. De nombreuses applications utilisent à présent *CCS* comme un moyen de spécifier des systèmes parallèles communiquants, de les vérifier formellement, et même de générer des sous-ensembles logiciels [63].

Le traitement des langages est traditionnellement décomposé en trois grandes phases : l'analyse lexicale, qui consiste à reconnaître les mots de base (lexèmes), l'analyse syntaxique qui identifie les relations structurelles caractérisant les combinaisons licites de lexèmes, et le traitement sémantique qui se décompose lui-même en analyses sémantiques et en générations de code cible. Nous suivrons ce schéma traditionnel en décrivant les différentes techniques impliquées dans ces traitements, même s'il comporte une connotation très "séquentielle" qui ne reflète pas de manière générale la complexité de leur intrications mutuelles.

[1] Posséder des critères de comparaison est une des arcanes de la connaissance, dans la mesure où la comparaison entraîne la *différenciation*, qui elle-même précède les activités de synthèse par recombinaisons.

```
I :=1; while I<=5 do I := I + 1
```

FIG. 3.1 : Un programme texuel élémentaire réalisant cinq itérations

Au préalable, il est intéressant de positionner l'ensemble des traitements linguistiques dans la perspective des *structures*. Toute *information* devient *connaissance* lorsqu'elle revêt une structure, c'est à dire une forme pouvant se réduire à un schéma connu, ce qui lui permet de devenir *opérationnelle* dans un système de traitement. C'est un fait bien connu qu'un algorithme s'applique généralement à des données possédant la structure adéquate. Trouver la forme adaptée, c'est réaliser la moitié du chemin conduisant à la solution. Plus on encode de sémantique dans un système de traitement, plus sa structure doit être riche et adaptée. Ainsi, les différentes phases du traitement des langages textuels correspondent à la manipulation et à des transformations structurelles de plus en plus riches. La sémantique est présente à tous les niveaux : elle passe seulement d'un état implicite à un état explicite, en changeant de forme, grâce à une structure s'enrichissant progressivement. Dans les langages textuels, les structures du niveau lexicales sont linéaires, celles du niveau syntaxique sont arborescentes, et celles du niveau sémantique sont des graphes. L'exemple de la figure 3.2 illustre le traitement d'un programme textuel élémentaire, réalisant une itération simple . La structure d'origine est une succession de caractères alphabétiques et d'espaces. La sémantique de ce programme est entièrement contenue dans la chaîne. L'analyse lexicale fait apparaître le séquencement des lexèmes et les informations associées. L'analyse syntaxique construit un arbre reflétant la grammaire du langage. L'analyse sémantique élabore un graphe reflétant l'exécution temporelle (chaque flèche représente l'écoulement du temps).

Dans un langage visuel, la richesse du système de représentation implique l'utilisation de graphes dès le niveau lexical. Cependant, le processus de transformation est de nature identique, entraînant des transformations structurelles qui permettent d'adapter la structure aux traitements requis. Les langages visuels nous apprennent toutefois que ces transformations structurelles ne vont pas forcément du plus simple au plus compliqué. L'exemple de la figure 3.3 montre le traitement linguistique d'un programme visuel qui est l'équivalent de l'itération de la figure 3.1. La représentation syntaxique utilise un graphe dont certains noeuds symbolisent les relations spatiales des objets graphiques ("*Contient*", "*au-dessus-de*", "*à-gauche-de*"). Ici, il est évident que les transformations réduisent la complexité initiale vers une structure plus simple, mais mieux adapté à la représentation de sa sémantique.

3.2.2 Traitements lexicaux

En considérant l'échelle d'expressivité des langages, les traitements lexicaux se situent au niveau élémentaire. Leurs principales fonctionnalités sont l'analyse lexicale et la transduction (principalement utilisée dans le traitement des langues naturelles). Leurs fondements théoriques sont les expressions régulières, caractérisant l'ensemble des mots constituant le langage lexical et les automates finis, autorisant la reconnaissance ou la transduction du langage. L'analyse lexicale est la première étape simplificatrice dans le traitement des langages, en transformant une suite de caractères en une séquence d'éléments qui sont les mots du langage. Ces mots ont différents types qui doivent être reconnus par l'analyseur lexical et encodés afin de faciliter les traitements de niveaux supérieurs. Les mots n'appartenant pas au langage doivent être reconnus comme erronés et signalés par un message. Le traitement de l'erreur varie en fonction des choix liés au langage : récupération de l'erreur et continuation ou arrêt du traitement.

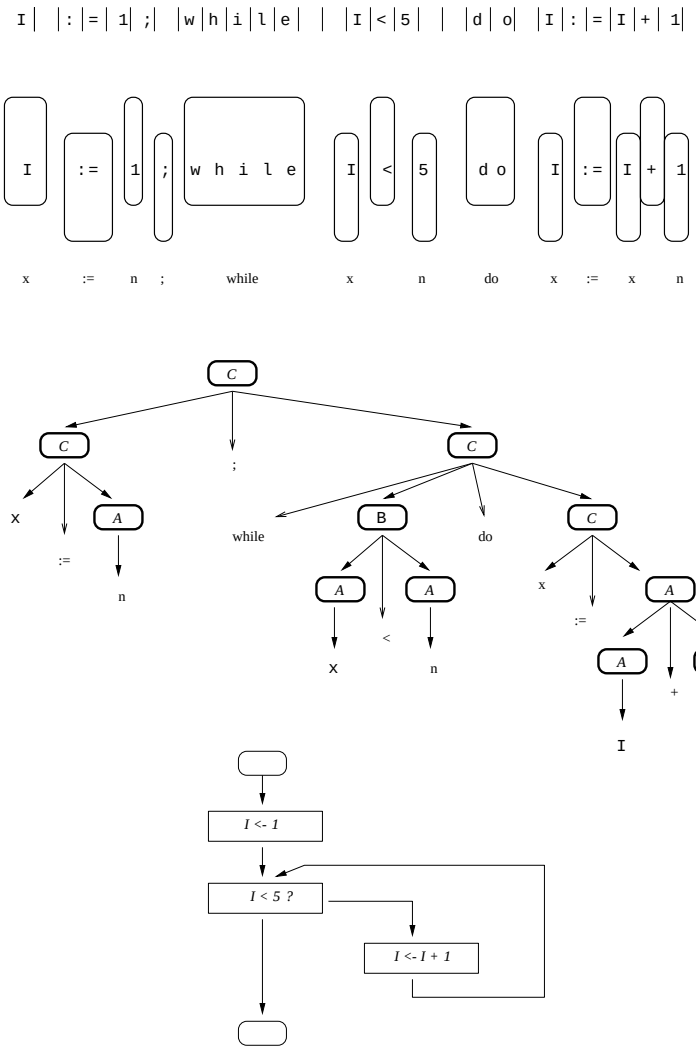


FIG. 3.2 : Transformation structurelle du programme

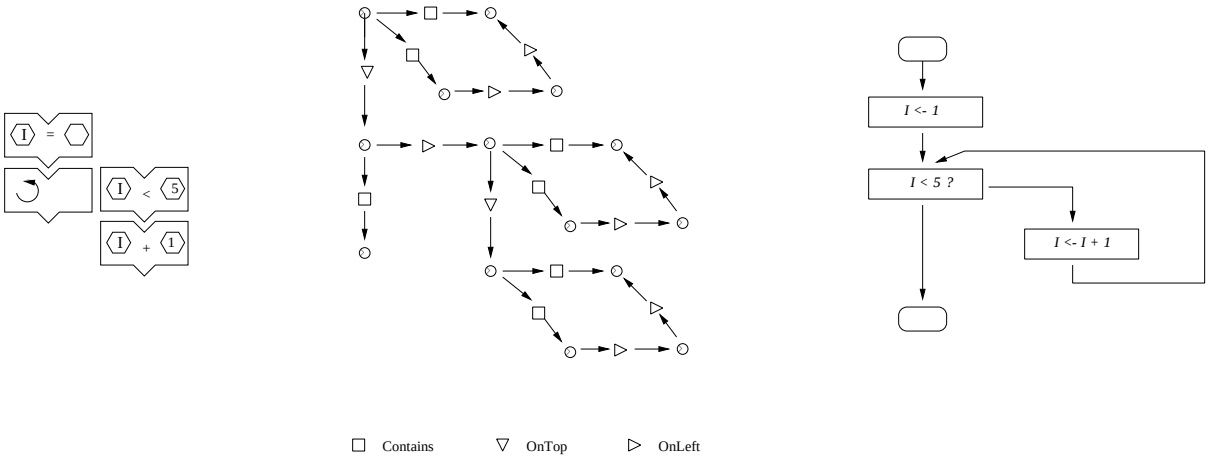


FIG. 3.3 : Un programme visuel, sa représentation syntaxique, et sa représentation sémantique

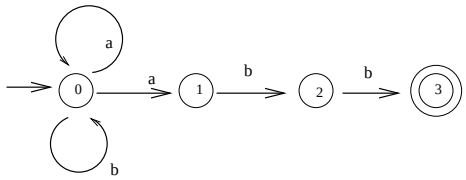


FIG. 3.4 : Automate fini non déterministe équivalent à $(a|b)^*abb$

expressions régulières

Les expressions régulières ont été étudiées par Kleene en 1956 qui s'intéressait à décrire les modèles d'activité nerveuse basée sur les automates à états finis de Mc Culloch et Pitts. Si les modèles neuronaux ont dû attendre le regain d'intérêt récent sur les systèmes neuromimétiques, les travaux de Kleene ont rapidement fertilisé l'étude formelle de l'analyse lexicale puis de la génération automatique d'analyseurs lexicaux. Des expressions régulières permettent de spécifier une classe de langages obtenus par des opérations simples sur un ensemble d'éléments de base appelé alphabet. Un élément spécial de l'alphabet, ϵ, est neutre pour l'opération de concaténation des chaînes de caractères. Pour un alphabet $\mathcal{A} = \{a, b\}$, on note $\mathcal{A}^0$ le langage $\{\epsilon\}$, $\mathcal{A}^1 = \{a, b\}$, $\mathcal{A}^2 = \{aa, ab, bb, ba\}$. D'une manière générale, on note $\mathcal{A}^i$ le langage obtenu par i concaténations de n'importe quelle lettre de l'alphabet $\mathcal{A}$. La notation $\mathcal{A}^*$ est la fermeture de Kleene, qui peut aussi s'écrire $\bigcup_{i\geq 0} \mathcal{A}^i$. La notation $\mathcal{A}^+$ est la fermeture positive, qui peut aussi s'écrire $\bigcup_{i\geq 1} \mathcal{A}^i$. La construction récurrente s'écrit simplement $\mathcal{A}^i = \mathcal{A}^{i-1}\mathcal{A}$. Sur cette définition des langages de niveau lexical, on peut introduire les expressions régulières. Si on note r, s des expression régulières quelconques, on appelle $L(r), L(s)$ les langages associés. La table suivante définie les opérateurs du langage :

$r\|s$	$L(r) \cup L(s)$
rs	$L(r)L(s)$
r^*	$L(r)^*$
r^+	$L(r)^+$

Les langages dénotés par une expression régulière sont appelés "ensembles réguliers". Ces langages ont une expressivité limitée. Par exemple, l'ensemble $\{wcw\}$, où w représente une chaîne donnée, ne peut pas être décrit par une expression régulière.

automates de reconnaissance

Le problème consistant à reconnaître un langage dénoté par une expression régulière quelconque peut être résolu au moyen d'automates finis, déterministes ou non déterministes. Ce sont des ensembles d'états et de transitions entre états qui se produisent sur chaque lettre lue dans la chaîne à reconnaître. L'automate de la figure 3.4 reconnait le langage engendré par $(a|b)^*abb$. Un mot du langage place l'automate sur un état terminal lorsque toutes les lettres sont lues. Dans le cas contraire, le mot est rejeté. L'exemple cité est un automate indéterministe car deux transitions portant le même label sortent du même état : l'indéterminisme vient que pour une lettre donnée (ici *a*) , le choix de la transition est aléatoire. Toute expression régulière peut être transformée en un automate fini indéterministe équivalent et réciproquement. Il existe également un algorithme (décrit dans [4]) permettant de transformer un automate indéterministe en son équivalent déterministe. Ce dernier offre une mise en œuvre plus efficace mais

parfois au prix d'un plus grand nombre d'états et de transitions. Un algorithme découvert ultérieurement, permet de générer directement des automates finis déterministes à partir d'expressions régulières. La transformation d'une spécification de haut niveau (expression régulière) en une représentation de bas niveau (automate fini déterministe) est représentative des traitements liés à la compilation. Typiquement, chaque structure intermédiaire ainsi que la transformation permettant de l'élaborer est l'élément connu de la chaîne de résolution. Un problème difficile est traité plus aisément en identifiant des étapes solubles, ou déjà résolues dans un autre contexte. On trouve dans [4] les détails nécessaires à l'implantation de ces algorithmes.

Générateurs d'analyseurs lexicaux

Le prototype de ce genre d'outils est *Lex*. Ce dernier génère un programme C à partir d'une collection d'expressions régulières décrivant les lexèmes du langage. Les expressions sont traduites en un automate non déterministe unique. Des actions, spécifiées en langage C peuvent être exécutées lorsque l'analyseur reconnait une unité lexicale associée. Ces actions concernent également le contrôle de l'exécution : un identificateur symbolique est éventuellement retourné au programme exploitant l'analyse, comme un analyseur syntaxique. Deux paramètres importants pour la généricité de l'outil sont donc confondus au niveau de la spécification : la dépendance au langage d'implantation et la dépendance au schéma d'exécution lié à l'utilisation du noyau de traitement généré. La désambiguation des unités lexicales utilise la priorité donnée par l'ordre de déclaration des expressions régulières.

3.2.3 Traitements syntaxiques

Les traitements syntaxiques se situent au second niveau de l'échelle d'expressivité. Leurs principales fonctionnalités sont l'analyse syntaxique et le formatage de textes sources (*pretty-printing*). Leurs fondements théoriques sont les grammaires *hors-contexte*. Cependant, des développements assez récents permettent d'envisager plus largement les traitements syntaxiques, dans le domaine des langages visuels. Dès lors, les grammaires associées sont *positionnelles*, *relationnelles*, *grammaires de graphes*. Parallèlement à l'analyse lexicale, l'analyse syntaxique textuelle transforme une séquence de lexèmes en un arbre dont la structure reflète la grammaire qui régie l'organisation des mots du langage. De même, ce traitement doit être capable de discriminer les enchaînements illicites et doit offrir des mécanismes de recouvrement d'erreurs dépendant de l'implantation de l'analyseur. Pour les grammaires plus élaborées, l'arbre est remplacé par un graphe, ou plus richement décoré.

Grammaires hors-contexte

Les grammaires formelles ont été introduites par Chomsky en 1956 dans le cadre de l'étude des langues naturelles, et classifiées en quatres grandes familles du plus général au plus particulier : les grammaires syntagmatiques, sous-contexte, hors-contexte et grammaires d'états fini (ou grammaires de Kleene) qui sont équivalentes aux expressions régulières. Elles furent indépendamment adaptées par Backus qui travaillait sur Algol 60. Le principe de définition inductive, c'est à dire utilisant une fonction récurrente pour définir un ensemble, était déjà utilisé en mathématiques. Avec l'introduction des grammaires formelles, c'est l'ensemble de la théorie du traitement des langages qui prend corps, emmenée par des pionniers tels que Knuth, Wirth, Hoare, et bien d'autres. Leur importance vient de ce qu'elles permettent de décrire des ensembles infinis et riches au moyen d'un jeu réduit de motifs structuraux. De

$$
\begin{array}{lcll}
C & \rightarrow & x \text{ ':=' A} & (1)\\
C & \rightarrow & \text{skip} & (2)\\
C & \rightarrow & C \text{ ';' } C & (3)\\
C & \rightarrow & \text{'if' } B \text{ 'then' } C \text{ 'else' } C & (4)\\
C & \rightarrow & \text{'while' } B \text{ 'do' } C & (5)\\
B & \rightarrow & \text{'true'} & (6)\\
B & \rightarrow & \text{'false'} & (7)\\
B & \rightarrow & \text{'not' } B & (8)\\
B & \rightarrow & B \text{ 'and' } B & (9)\\
B & \rightarrow & A \text{ '}\leq\text{' } A & (10)\\
B & \rightarrow & A \text{ '=' } A & (11)\\
A & \rightarrow & n & (12)\\
A & \rightarrow & x & (13)\\
A & \rightarrow & A \text{ '+' A} & (14)\\
A & \rightarrow & A \text{ '-' A} & (15)\\
A & \rightarrow & A \text{ '*' A} & (16)
\end{array}
$$

FIG. 3.5 : La grammaire hors-contexte du langage *While*.

plus, l'hypothèse sémantique [1] permettant de relier les règles de grammaire aux ensembles décrits est très simple, et donc facile à comprendre et à manipuler. L'étude des grammaires formelles reste encore très active , dans le domaine des langages naturels, de l'intelligence artificielle et même dans le domaine des systèmes concurrents [42]. Une grammaire hors-contexte est décrite par un ensemble de règles ou productions. Chaque production comprend une partie gauche, unique, appelée *non-terminale* et une partie droite constituée d'un nombre quelconque de *non-terminaux* ou de *terminaux*. Ces derniers sont des unités lexicales, ou plus précisément, les types de ces unités lexicales. Le vocable "*terminaux*" exprime le fait que ces éléments ne peuvent être réécrits, et représentent donc une étape terminale du processus de production du langage. Un non-terminal particulier, nommé *axiome*, est universellement impliqué quelque soit la phrase appartenant au langage engendré par la grammaire. La forte puissance expressive des grammaires formelles provient du fait qu'un non-terminal peut être impliqué simultanément à droite *et* à gauche d'une règle, c'est à dire qu'une grammaire peut capturer des schéma de récursion complexes et virtuellement infinis.

Une grammaire peut être appréhendée de deux manières opposées : soit comme un système permettant de générer un ensemble de phrases (langage de niveau syntaxique), soit comme un système permettant de reconnaître le même langage. Dans la première approche, la génération se fait en partant de l'axiome, et en utilisant toute production applicable à un non-terminal de la partie droite pour réécrire ce dernier. Le processus s'arrête lorsque tout non-terminal a été substitué : la phrase obtenue ne comprend alors que des unités lexicales. La figure 3.5 présente la grammaire spécifiant la syntaxe d'un langage d'étude, nommé *While*, de type impératif, utilisé entre autre par John Hannan [57] pour étudier la génération de compilateurs à partir de leurs sémantiques opérationnelles.

En utilisant cette grammaire de manière générative, la phrase de la figure 3.1 peut-être obtenue par la chaîne de dérivation suivante :

[1] La réécriture des non-terminaux.

$$
\begin{array}{rcll}
C & \Rightarrow & C \text{ ';' } C & (3)\\
 & \Rightarrow & x \text{ ':=' } A \text{ ';' } C & (1)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' } C & (12)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } B \text{ 'do' } C & (12)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } A \text{ '}\leq\text{' } A \text{ 'do' } C & (10)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } A \text{ 'do' } C & (13)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } n \text{ 'do' } C & (12)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } n \text{ 'do' } x \text{ ':=' } A & (1)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } n \text{ 'do' } x \text{ ':=' } A \text{ '+' } A & (14)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } n \text{ 'do' } x \text{ ':=' } x \text{ '+' } A & (13)\\
 & \Rightarrow & x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } n \text{ 'do' } x \text{ ':=' } x \text{ '+' } n & (12)
\end{array}
$$

Les numéros placés à droite référencent les règles de grammaires appliquées. L'exemple développe une dérivation *gauche*, en ce sens que la réécriture s'applique systèmatiquement au non-terminal situé le plus à gauche. Conventionnellement, cet ensemble de réécritures se note :

$$C \quad \stackrel{*}{\Rightarrow} \quad x \text{ ':=' } n \text{ ';' 'while' } x \text{ '}\leq\text{' } n \text{ 'do' } x \text{ ':=' } x \text{ '+' } n$$

où l'opérateur $*$ placé sur $\Rightarrow$ a le même sens que l'opérateur de Kleene des expressions régulières (d'ailleurs, on peut également utiliser l'opérateur de fermeture positive $+$ pour exprimer que la chaine terminale est dérivée en une ou plusieurs étapes).

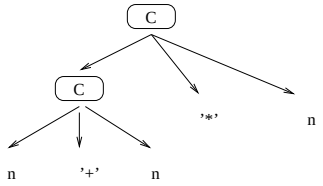
La deuxième approche est analytique et consiste à reconnaître les phrases du langage en tentant de les ramener aux structures grammaticales. La figure 3.2 montre l'arbre syntaxique construit en analysant la même phrase que précédemment. L'implantation d'analyseur est une tâche difficile, qui peut être automatisée de différentes manières en fonction des propriétés de la grammaire considérée [4]. Un des problèmes récurrent en analyse syntaxique est la résolution des ambiguités : les règles

$$
\begin{array}{rcll}
C & \rightarrow & n \text{ '+' } n & (1)\\
C & \rightarrow & n \text{ '*' } n & (2)
\end{array}
$$

introduisent une ambiguité lors de la reconnaissance : l'analyse de $1 + 2 * 3$ par la réduction de (1) suivie de (2)

$$
\begin{array}{rcrl}
n \text{ '+' } n \text{ '*' } n & \Rightarrow & C \text{ '*' } n & (1)\\
 & \Rightarrow & C & (2)
\end{array}
$$

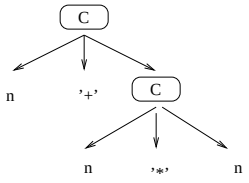
conduit à l'arbre syntaxique



et la réduction de (2) suivie de (1)

$$
\begin{array}{rcrl}
n \text{ '+' } n \text{ '*' } n & \Rightarrow & n \text{ '+' } C & (2)\\
 & \Rightarrow & C & (1)
\end{array}
$$

produit l'arbre



Bien évidemment, cette divergence induit une ambiguité sémantique, puisque le résultat du calcul sera ou 9, ou 7.

D'une manière générale, les grammaires hors-contexte sont catégorisées en une hiérarchie reflétant leur expressivité. Les algorithmes applicables au problème de reconnaissance constituent le critère de classification. Ils seront abordés plus en détail en 3.2.3.

Les langages décrits par des grammaires hors-contexte sont plus généraux que ceux décrits par des expressions régulières. Notamment, les emboîtements de parenthèses servant à décrire des expressions arithmétiques sont correctement spécifiés par les grammaires mais ne peuvent être capturés dans une expression régulière. Cependant, les langages de programmation "réels" possèdent une expressivité non couverte entièrement par ce type de grammaire. Les constructions telles que les déclarations de variables suivies de leur utilisation nécessitent une analyse contextuelle et donc un niveau sémantique supérieur. Par exemple, déterminer si une variable référencée dans une expression a été préalablement déclarée nécessite de consulter un dictionnaire dont les entrées sont les noms des variables déclarées (le contenu est généralement le type associé). Ce contrôle simple et courant ne peut pas être implanté directement dans la structure grammaticale.

Grammaires multi-dimensionnelles

Dans une grammaire "classique", les éléments décrits dans les parties droites des règles sont implicitement ordonnés par une relation de précédence. Ce qui parait une évidence dans l'approche séquentielle ordonnée inhérente aux langages textuels, transcription écrite de la parole elle-même strictement ordonnée dans le temps, peut être reconsidérée dans le cadre de langages visuels possédant plus d'une dimension expressive. Est-il possible d'identifier et d'encoder des structures syntaxiques régissant les relations spatiales, morphologiques, chromatiques dans un alphabet de lexèmes visuels ? Plusieurs approches ont été explorées et caractérisées, bien que des unifications restent à proposer. Nous citerons tout d'abord les grammaires positionnelles [38], les grammaires à contraintes relationnelles [122], les grammaires de relations [49], les grammaires de graphes et d'hypergraphes [39, 94, 48]. Pour les trois premières approches, le principe est de spécifier explicitement les relations unissant les termes droits des règles de production. De plus, des attributs doivent être associés aux terminaux et non terminaux, afin d'exprimer les informations supplémentaires permettant de caractériser les éléments. Typiquement, ces attributs sont la position x, y, ou même z lorsque les trois dimensions sont prises en compte. Ainsi, les grammaires positionnelles sont considérées comme la généralisation dans l'espace des grammaires hors-contexte classiques. Les productions prennent la forme suivante :

$$A \quad \rightarrow \quad x_1 R_1 x_2 \ldots R_{m-1} x_m, \quad \Delta$$

où les x_i sont des symboles terminaux ou non terminaux, les R_i sont des relations composites unissant des symboles d'indice 0 à i, Δ est une fonction permettant de synthétiser les attributs de A en fonction des attributs des x_i. L'exemple suivant définit la syntaxe du langage visuel iconique mis en œuvre dans la figure 2.3 de la section 2, qui présentait un programme spécifié en *Vampire*. Des icones "*carré*" (lexème c), contenant un losange (lexème l) ou une patatoïde (p) sont juxtaposées soit à droite d'une autre icone, soit en dessous. Les attributs spécifiant la position d'une icone i sont x et y, notés i_x et i_y. Les relations spatiales sont *Hor*, définie par

$$i \textit{ Hor } j \Leftrightarrow (i_x \leq j_x) \textit{ and } (i_y = j_y)$$

et *Ver* définie par

$$i \textit{ Ver } j \Leftrightarrow (i_y \leq j_y) \textit{ and } (i_x = j_x).$$

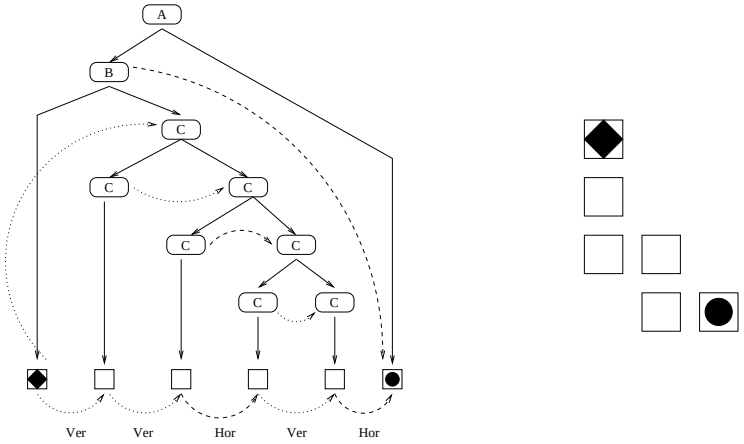
La grammaire positionelle (simplifiée) $\mathcal{PG}$ est

$$
\begin{array}{lcll}
A^0 & \rightarrow & B^1 \textit{ Hor } p^2 & \{{A^0}_x = {p^2}_x, {A^0}_y = {p^2}_y\} \quad (1)\\
A^0 & \rightarrow & B^1 \textit{ Ver } p^2 & \{{A^0}_x = {p^2}_x, {A^0}_y = {p^2}_y\} \quad (2)\\
B^0 & \rightarrow & l^1 \textit{ Hor } C^2 & \{{B^0}_x = {C^2}_x, {B^0}_y = {C^2}_y\} \quad (3)\\
B^0 & \rightarrow & l^1 \textit{ Ver } C^2 & \{{B^0}_x = {C^2}_x, {B^0}_y = {C^2}_y\} \quad (4)\\
C^0 & \rightarrow & C^1 \textit{ Hor } C^2 & \{{C^0}_x = {C^2}_x, {C^0}_y = {C^2}_y\} \quad (5)\\
C^0 & \rightarrow & C^1 \textit{ Ver } C^2 & \{{C^0}_x = {C^2}_x, {C^0}_y = {C^2}_y\} \quad (6)\\
C^0 & \rightarrow & c^1 & \{{C^0}_x = {c^1}_x, {C^0}_y = {c^1}_y\} \quad (7)
\end{array}
$$

En utilisant cette syntaxe, on montre que la phrase de l'exemple 2.3 appartient à $\mathcal{L}(\mathcal{PG})$, langage engendré par $\mathcal{PG}$ en produisant une dérivation gauche

$$
\begin{array}{lcll}
A & \Rightarrow & B \textit{ Hor } p & (1)\\
 & \Rightarrow & l \textit{ Ver } C \textit{ Hor } p & (4)\\
 & \Rightarrow & l \textit{ Ver } C \textit{ Ver } C \textit{ Hor } p & (6)\\
 & \Rightarrow & l \textit{ Ver } c \textit{ Ver } C \textit{ Hor } p & (7)\\
 & \Rightarrow & l \textit{ Ver } c \textit{ Ver } C \textit{ Hor } C \textit{ Hor } p & (5)\\
 & \Rightarrow & l \textit{ Ver } c \textit{ Ver } c \textit{ Hor } C \textit{ Hor } p & (7)\\
 & \Rightarrow & l \textit{ Ver } c \textit{ Ver } c \textit{ Hor } C \textit{ Ver } C \textit{ Hor } p & (6)\\
 & \Rightarrow & l \textit{ Ver } c \textit{ Ver } c \textit{ Hor } c \textit{ Ver } C \textit{ Hor } p & (7)\\
 & \Rightarrow & l \textit{ Ver } c \textit{ Ver } c \textit{ Hor } c \textit{ Ver } c \textit{ Hor } p & (7)
\end{array}
$$

Le lecteur peut remarquer que cette syntaxe exprime la contrainte que toute chaîne doit commencer par un losange et se terminer par une patatoïde. Comme la sémantique de l'exemple le montre, toute chaîne de $\mathcal{L}(\mathcal{PG})$ est donc entièrement évaluée sur la base des règles qui définissent la propagation du contrôle de la gauche vers la droite et du haut vers le bas. La définition des relations Hor et Ver n'impose aucune contrainte sur la distance qui sépare les icônes, et donc, des chaînes disjointes peuvent être reconnues dès lors qu'elles respectent les relations spatiales d'alignement. Le schéma suivant montre l'arbre syntaxique que l'on peut associer à l'analyse de cette phrase.



Cet exemple appelle plusieurs commentaires. D'une part, la structure syntaxique reste arborescente ce qui est un avantage simplificateur concernant sa manipulation. C'est la conséquence directe de la morphologie des règles, qui suit *grosso modo* la structure hors-contexte classique. La puissance expressive est considérablement augmentée par rapport à une grammaire linéaire : les fonctions relationnelles sont variées, prenant en compte des paramètres tels que couleurs, distance, orientation, ou d'autres attributs plus abstraits. Cependant, les notations utilisées pour décrire la grammaire sont beaucoup plus complexes, reflètant la richesse du système de représentation. De plus, les relations n'étant pas directement codées dans la topologie de la structure (il faudrait pour cela utiliser un graphe), elles doivent être spécifiées par ajout de fonctions qui alourdissent la manipulation et la compréhension des productions. C'est une remarque tout à fait générale que la manipulation de systèmes de représentation très riches requière un formalisme et des notations proportionnellement complexes. L'étude épistémologique des

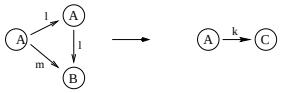
mathématiques a démontré l'influence décisive des sytèmes de notations (donc de représentation !) dans l'évolution des concepts. Dans un très intéressant article, Leslie Lamport [70] écrit [1] :

> *Mathematical notation has improved over the past few centuries. In the seventeenth century, a mathematician might have written*
>
> > There do not exist four positive integers, the last being greater than two, such that the sum of the first two, each raised to the power of the fourth, equals the third raised to that same power.
>
> *How much easier it is to read the modern version*
>
> *There do not exist positive integers* x, y, z, *and* n, *with* $n > 2$, *such that* $x^n + y^n = z^n$

Une solution "naturelle" semble être d'utiliser des méta-langages visuels pour formaliser les langages visuels. Cependant, même si cette voie semble être généralement empruntée par les chercheurs du domaine, elle comporte des inconvénients : il est difficile d'abstraire certains concepts par rapport à d'autres, précisement à cause du caractère extensionnel et global des systèmes visuels, comme évoqué dans le chapitre 2. Dans *Vampire* [76], lorsque le développeur de langage veut définir une règle sémantique portant sur un attribut non spatial (tel que la couleur), il doit utiliser un symbole particulier pour exprimer que la position de l'icône en partie gauche ne doit pas être prise en compte pour le déclenchement de la règle.

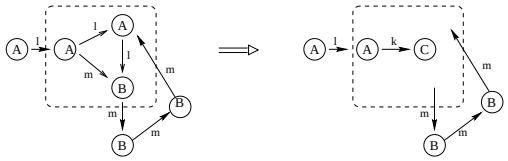
Les grammaires de relations [49, 114] ressemblent de près aux grammaires positionnelles mais sont plus générales car les relations imposées aux membres droits ne sont pas seulement binaires mais n-aires. De plus, les productions sont divisées en deux ensembles distincts : les règles *structurales* (*s-items productions*) qui sont désignées par un nom unique et les règles relationnelles (*r-items productions*). Les deux systèmes de règles sont mis en jeu lors de l'application d'une production : les grammaires de relation réécrivent aussi bien les symboles que les relations entre ces symboles. Leur schéma d'exécution est de ce fait beaucoup plus complexe. Nous entrons à présent dans un formalisme plus riche et plus pointu. C'est malheureusement le prix qu'il faut payer pour accéder à la plus puissante évolution proposée pour les grammaires "classiques", possèdant une "colonne vertébrale" hors-contexte (*context-free backbone*). Ces travaux assez récents ne sont toutefois pas présentés en détail, car cette classe de grammaires demande des algorithmes d'analyse difficiles à implanter et la complexité de leurs spécifications limite leur utilisation dans le cadre d'outils généraux. Il semble qu'un certain travail théorique reste à réaliser afin de définir plus simplement la sémantique des productions (comparé par exemple aux morphismes des grammaires de graphe algébriques [48])

Dans les grammaires de graphes, les productions décrivent des sous-graphes en partie gauche et droite, c'est à dire des ensembles d'éléments (*nœuds*) et d'arcs qui décrivent leurs relations. Ces dernières sont donc *explicites*, en opposition aux relations implicites des grammaires positionnelles ou relationnelles, exprimées au moyen de fonctions sur les attributs des nœuds. Le concept est donc plus intuitif et plus cohérent puisque les nœuds et les arcs sont réécrits simultanément, avec le même formalisme. Le principal problème réside dans la connexité du graphe après application de la règle. Si le sous-graphe en partie gauche possède un certain nombre d'arcs entrants et sortants qui le relient avec son contexte dans le graphe d'application, il est possible que la dérivation produise des graphes incohérents, possèdant des arcs non reliés. Par exemple, la règle



[1] A propos de la conjecture de Fermat, de toute évidence. Celle-ci a été prouvée récemment dans une démonstration de 50 pages, contenant d'ailleurs quelques erreurs lors de sa première soumission.

s'appliquant strictement au graphe suivant



produit des arcs "pendants", car non connectés (problème des *dangling edges*). Un certain nombre de conditions sont donc associées à l'application des règles, ou des actions implicites telles que la destruction des arcs incohérents sont appliquées de manière systématique.

Dans l'approche algébrique ([48]) la condition d'application d'une production est appellée *condition d'adhérence* (*gluing condition*), et preserve la connexité lors de la réécriture. Plus précisément, une règle de production p est une paire de graphes $\langle L, R\rangle$ appellés partie gauche et droite, dont on peut extraire un sous-graphe gauche $K_L \subseteq L$ et un sous-graphe droit $K_R \subseteq R$ tels que K_L et K_R soient en correspondance bi-univoque (bijection) [1]. Concrètement, cela signifie qu'il doit exister un noyau commun qui n'est pas transformé lors de la réécriture. Une règle p peut être appliquée à un graphe G si :

1. L est un sous-graphe de G. Plus précisément, il doit exister un morphisme g de L sur G permettant d'établir cette inclusion dans G. Formellement, un morphisme f est une application d'un ensemble E muni d'un opérateur $\Box$, vers un ensemble F muni d'un opérateur $\triangle$ tel que

   $$\forall x, y \in E, f(x \Box y) = f(x) \triangle f(y)$$

   Intuitivement, celà signifie que si un ensemble source est structuré par un opérateur, la transformation de cet ensemble par f transpose cette structure dans F. Dans le cas des graphes, les opérateurs impliqués dans le morphisme sont les relations qui définissent les arcs.

2. p vérifie la condition d'adhérence (*gluing condition*). Cette condition s'exprime sur l'ensemble A des nœuds de K_L (nœuds adhérents), l'ensemble P des nœuds pendants (*dangling edges*, les nœuds de $P \subseteq L$ tels que les nœuds de $g(P)$ soient la cible ou la source d'arcs de $G - g(L)$), et l'ensemble $I \subseteq L$ des nœuds identifiés dans $g(L)$, $I = \{x, y \in L, x \neq y \mid g(x) = g(y)\}$:

   $$P \cup I \subseteq A$$

   Cette contrainte, qui porte uniquement sur les nœuds du graphe, évite de manipuler des conditions supplémentaires concernant l'emboîtement.

Une dérivation se déroule alors en trois phases :

1. Identification de la partie gauche avec un sous-graphe de G. Le morphisme g doit être établi, et la condition d'adhérence vérifiée. Dans le cas contraire, la production n'est pas appliquée.
2. Effacement de $g(L - K_L)$, qui produit un graphe $D = G - [g(L - K_L)]$
3. Création du graphe final $H = D \oplus R$. Ici, l'opérateur $\oplus$ réalise le "collage", c'est à dire l'ajout des nouveaux nœuds et des arcs associés.

Les développements algébriques utilisent des combinaisons de morphismes appelés *pushouts*, et conduisent à d'élégants et rigoureux résultats sur les propriétés de réécriture des graphes. Notamment, il est possible d'amalgamer deux productions en une seule sous réserve que les graphes d'adhérence et d'effacement présentent certains recouvrements.

[1] Ils ne peuvent être considérés comme simplement identique, car formellement, ils n'appartiennent pas aux mêmes graphes

Analyseurs syntaxiques

Un analyseur syntaxique réalise la reconnaissance d'une phrase d'un langage engendré par une grammaire et construit généralement une structure associée. Pour les grammaires hors-contexte, il est possible de générer automatiquement un certain nombre d'analyseurs dont le comportement et la puissance varient en fonction des propriétés de la grammaire. Pour les grammaires multi-dimensionnelles, des analyseurs généraux ont étés proposés, avec des performances linéaires, logarithmiques et même exponentielles en fonction de l'expressivité des grammaires et des modèles de graphe. C'est un domaine de recherche actif dont les applications sont assez générales, des langages visuels à la reconnaissance d'image. Reckers et Shürr [94, 95] produisent une comparaison qualitative des différentes approches.

On distingue deux grandes familles d'analyse : l'analyse descendante, qui exploite les informations grammaticales pour diriger l'analyse de la phrase, et l'analyse ascendante, qui progresse en lisant les lexèmes de la phrase et construit la structure en "montant" dans les règles.

Les grammaires de classe LL(1) permettent de construire des analyseurs prédictifs efficaces (descendants). La compilation de la grammaire calcule une table, parcourue par l'analyseur au moyen d'un index sur le non-terminal courant et un index sur l'unité lexicale courante. Les grammaires LL(1) (*Left to right scanning, Left most dérivation, 1 prevision symbol*) vérifient certaines propriétés permettant d'obtenir une table sans entrées multiples. Notamment, la grammaire ne doit pas comporter de règles récursives à gauche, du type $A \rightarrow AB$. Des algorithmes permettent, autant que possible, d'éliminer de tels schémas en transformant la grammaire d'origine.

Les grammaires de classe LR (*Left to right scanning*) permettent de générer des analyseurs ascendants efficaces, et plus généraux que les grammaires LL. En fonction des propriétés de la grammaire, les analyseurs générés sont dans l'ordre de puissance, SLR (*Simple LR*), LALR (*LookAhead LR*) ou LR canonique. Si les performances à l'exécution sont comparables, les coûts de compilation et la dimension des tables sont très différents. Des algorithmes de compression rendent leurs utilisation réaliste, et la plupart des langages de programmation utilisent ces techniques. Pour tous les analyseurs présentés, la grammaire ne doit pas être ambiguë, c'est à dire offrir plusieurs possibilités de réduction pour la même séquence d'entrée. Cependant, il est utile pratiquement de conserver ces ambiguités lorsqu'elles représentent correctement les ambiguités du langage, comme l'associativité et les priorités sur les opérateurs. Les outils de génération d'analyseurs proposent donc des mécanismes pour résoudre les conflits d'analyses, car ils permettent de conserver des grammaires plus simples et plus naturelles.

De nombreux travaux ont étés conduits sur l'analyse syntaxique incrémentale, capable de reconnaître une phrase en fragmentant l'analyse, technique bien adaptée aux outils d'édition "intelligents", dans lesquels la vérification syntaxique est réalisée en continu, ou de manière interactive. Ce type d'analyseur utilise les mêmes tables, mais implante un algorithme de reconnaissance plus complexe, capable de rétablir le contexte d'analyse en élaborant une structure plus riche que l'arbre syntaxique classique. J. M. Larchevêque [71] propose un algorithme optimal qui travaille sur des tables LALR(1) et utilise un arbre augmenté (*threaded tree*), qui permet de reconstruire la pile d'analyse par un parcours des nœuds. Hélas, cette approche n'est pas encore explorée lorsque le niveau syntaxique est franchi pour aller plus profondément dans le traitement sémantique [1]. Les difficultés sont liées essentiellement à l'architecture de ce dernier, aux effets de bords (utilisation de la mémoire globale), et aux schémas d'exécutions.

Les grammaires positionnelles que nous avons évoquées précédement peuvent être compilées et produire des analyseurs *pLR*, généralisation des analyseurs LR, présentant globalement la même efficacité

[1] Les grammaires attribuées offrent, dans une certaine mesure, la possibilité de traitements sémantiques incrémentaux. Toutefois, leur expressivité et leur sujettion aux schémas d'exécution (imposés par leurs évaluateurs) restreignent leur applicabilité aux traitements généraux que nous envisagons

(hormis les temps de traitements induits par l'évaluation des contraintes du membre droit, ainsi que la recherche du lexème successeur où *look-ahead scanning*). Cette approche est appliquée (très récemment) dans le générateur VLCC [38] avec des résultats convaincants. Les auteurs utilisent ce type d'analyseur pour traiter aussi bien les grammaires textuelles que visuelles. Ils définissent plusieurs classes de langages visuels traitables, tels que des langages à flot de données, à diagrammes (assemblages de symboles avec des lois d'interconnexion) et de type organigrammes structurés (*flowchart*) qui sont considérés comme difficiles à analyser [94, 123].

3.2.4 Traitement des types

Le traitement des types concerne aussi bien la phase de compilation que la phase d'exécution des programmes. Le but de ces traitements est de détecter les erreurs liées aux utilisations invalides des données, et des opérations sur ces données. Le contrôle des types repose sur un système décrivant la manière de définir des types complexes à partir de types de base, d'assigner des types aux variables, d'établir les lois de transformation des types dans les opérations, et encore, de décrire les transformations intermédiaires licites permettant de résoudre des conflits (coercition). Les langages de programmation dont le type de chaque expression est déterminable à la compilation sont dits *typés statiquement*. Lorsque certains types ne sont pas connus statiquement mais que leur cohérence est garantie, le langage est dit *fortement typé*. Cela signifie que le compilateur garantit une exécution sans erreur de types. Un langage typé statiquement présente de meilleures performances d'exécution, favorise la détection d'erreurs à la source, force le développeur à une plus grande discipline de programmation. En contre partie, il souffre d'une moins grande souplesse et d'une perte d'expressivité en restreignant prématurément l'applicabilité des fonctions, procédures ou méthodes associées aux données. Cette classe de langages peut être représentée par Pascal ou Modula-2.

Le polymorphisme représente un progrès très significatif amenant une plus grande souplesse dans le traitement, mais aussi dans l'utilisation de la notion de types dans la programmation [24]. Les langages comme Pascal sont *monomorphiques* en ce sens que toute variable est interprétée comme ayant un et un seul type. Les langages polymorphes permettent la définition de variables et opérations possédant plusieurs types ayant certaines propriétés en commun. On distingue le polymorphisme paramétrique, le polymorphisme spécifique, et le polymorphisme par inclusion. Dans le polymorphisme paramétrique, le type est explicitement ou implicitement représenté par une variable qui entre dans les déclarations et la définition des fonctions. Ce paramètre doit être instancié lors de l'utilisation de la variable ou des opérations associées. Généralement l' approche explicite est dite générique, comme les *generic packages* de Ada ou encore les *templates* de C++. Le langage *ML* est considéré comme un modèle de mise en œuvre du polymorphisme paramétrique, où les types sont explicitement déclarés, soit totalement, soit partiellement, ou bien encore purement omis. Dans ces deux derniers cas, le système de types possède un mécanisme d'inférence basé sur un algorithme d'unification capable de déduire le type le plus général, ou de déclarer une incohérence. Bien qu'extrêmement souple, *ML* reste fortement typé.

Le polymorphisme spécifique est généralement lié à l'utilisation d'opérateurs syntaxiquement identiques mais ayant une sémantique adaptée aux types concernés. C'est par exemple l'opérateur + pour réaliser indistinctement les concaténations sur des chaînes de caractères ou des concaténations sur des listes, comme dans le langage Python. Le polymorphisme par inclusion est apparu avec le sous-typage et l'héritage des langages à objets. Il repose sur des propriétés ensemblistes des types et plus précisement sur la notion d'inclusion : le type *réel* peut être considéré comme un sous-ensemble du type *numérique*, de même que le type *entier*. Dès lors, toute opération définie sur des variables *numériques* est applicable indistinctement sur des variables de types *entier* ou *réel*. Dans les langages à objets, le sous-typage per-

met de généraliser cette approche à des structures de données complexes : un objet de type *voiture* hérite des méthodes définies sur le super-type *véhicule*.

Les évolutions de la théorie des types tentent d'étendre la souplesse, l'expressivité et la puissance des systèmes de traitement des types [43, 1]. Certaines approches, telle que celle developpée par Oscar Nierstrasz ("*regular types for active objects*" [87]), tentent par exemple d'incorporer le comportement temporel des objets dans un système de types muni d'une relation de sous-typage. La vérification des types, en tant que sous-ensemble de la vérification sémantique, joue un rôle essentiel comme trait d'union entre le modèle du langage et le système de traitement.

Ils sont conçus pour être vérifiables, transparents, et améliorer la qualité et la performance des langages. La vérification assure que le système de types, rigoureusement formalisé, possède les propriétés escomptées au niveau même de sa conception théorique. La transparence est la qualité qui rend intuitif au programmeur l'application des mécanismes de typage. Les améliorations au niveau du langage sont la souplesse quant à l'évolution et la réutilisation des sources, la rigueur et la modularité dans la structuration et la spécification, la détection au plus tôt des erreurs et l'amélioration des performances d'exécution par la production d'un code de meilleur qualité.
Ces caractéristiques expliquent pourquoi le système de types est au centre de la conception d'un langage.

Systèmes de types formels

Les systèmes de types sont habituellement formalisés au moyen d'assertions logiques appelées *jugements* de la forme $\Gamma \vdash A$, qui signifient que le prédicat A est évalué dans le contexte Γ. Les assertions sur les types s'écrivent $\Gamma \vdash M : \tau$, où le fragment de programme M a pour type τ dans le contexte Γ. Ainsi, par exemple, $\phi \vdash true : boolean$ est vrai dans un contexte vide (ϕ) ou, si $\Gamma = \{x : int\}$ alors $\Gamma \vdash x + 1 : int$ est également vrai. $\phi \vdash true : boolean$ signifie ici que $true$ a pour type $boolean$ ou plus précisément, que l'assertion "$true$ a pour type $boolean$" est vraie quelque soit le contexte. De même, $\Gamma \vdash x + 1 : int$ signifie que l'expression $x + 1$ a pour type int si le contexte Γ permet d'affirmer que x a pour type int. Ainsi, tout jugement est soit valide, soit invalide. L'ensemble des jugements valides est structuré au moyen de règles d'inférence qui permettent d'établir la validité d'un jugement sur la base de la validité d'autres jugements. La règle

$$\frac{\Gamma \vdash x : int \quad \Gamma \vdash y : int}{\Gamma \vdash x + y : int}$$

signifie que $\Gamma \vdash x + y : int$ est valide à la condition que $\Gamma \vdash x : int$ et $\Gamma \vdash y : int$ soient tous les deux valides. Ces règles ont pour qualité d'être modulaires, claires et précises. Elles permettent d'établir la validité ou l'invalidité d'une expression de typage en produisant un arbre de preuves dont la racine est l'expression, les nœuds sont les règles appliquées et les feuilles, des jugements valides. L'ensemble de ces règles constitue le système de types formel. Certaines propriétés de ce système logique peuvent être vérifiées en vue d'une utilisation valide. Notamment, il peut être complet (toute assertion de typage doit pouvoir être prouvée valide ou invalide), cohérent (des contradictions ne doivent pas coexister au sein du système) et correct, c'est à dire représentatif de la sémantique des types (*soundness*). Surtout, il doit pouvoir conduire à établir un algorithme, de complexité maîtrisée, permettant de *décider* si un programme est correctement typé ou non.

Contrôle des types

Indépendament du système formel qui décrit *logiquement* le traitement des types, un contrôleur doit réaliser concrètement la vérification lors du traitement sémantique. Un système formel valide ne produit

pas nécessairement un bon contrôleur, c'est à dire *correct* (en accord avec la formalisation) et efficace (d'une complexité réaliste). Un critère de qualité est particulièrement difficile à décrire et établir : le traitement des erreurs et la précision du report au programmeur. D'une manière générale, une bonne gestion des erreurs nécessite de conserver plus d'information au cours du traitement, et de concevoir une architecture spécifiquement adaptée. Un certain nombre de contrôles sont réalisés à l'exécution, même dans les langages statiquement typés, ou dans des langages mixtes [2]. Les tests ou erreurs entrent alors généralement dans le modèle d'exception lié au langage, et le programmeur peut les prendre en compte dans ses traitements.

Types visuels

Ici encore, il n'y a pas de distinction fondamentale (sémantique) entre la notion de types textuels et visuels. Il est certain toutefois que la transposition de ces notions dans un système de représentation visuel n'est pas évident si l'on désire prendre en compte ses propriétés intrinsèques afin d'obtenir un résultat intuitif. Très peu de travaux se sont essayés à une veritable transposition. Beaucoup utilisent une notion de typage sous-jacente (non représentée) ou partiellement explicitée au moyen de textes associés aux représentations. Les abstractions obtenues correspondent aux types de bases des langages textuels (lignes, points, polygones) que l'on peut combiner pour obtenir des types structurés. Dans [118], nous décrivons une approche qui offre un polymorphisme géométrique (par inclusion), c'est à dire, qu'un certain nombre d'opérateurs spatiaux (déplacement, dilatation) s'appliquent indistinctement à tous les types visuels. Le résultat permet de gérer plus facilement les problèmes d'extensibilité (*scalability*) des représentations et offre les mêmes qualités de simplification du modèle de programmation que dans les langages textuels [1]. Toutefois, les différences profondes entre l'approche textuelle et visuelle favorisent la complémentarité, et il est probable que les aspects fortements *symboliques*, c'est à dire plus *abstraits* restent manipulés et spécifiés textuellement.

3.2.5 Traitements sémantiques

Les traitements syntaxiques ne peuvent couvrir la totalité des besoins. Une grande partie des informations relatives au code est stockée dans les noeuds de l'arbre syntaxique, sous forme d'attributs. Une petite partie s'est donc transformée en structure, et a été reconnue par l'analyseur, donc vérifiée. Quelle que soit la finalité des traitements sémantiques, ils réalisent à différents niveaux, avec une profondeur plus ou moins poussée et dans des ordres différents, la vérification de certaines propriétés, et la transformation de l'information.

Sémantique formelle

La sémantique, c'est la signification, le sens. Evidemment, en théorie des langages, mais aussi dans d'autres domaines comme la génétique, cela prend une connotation plus précise. On définit la signification d'un code en fonction de l'interprétation que celui-ci peut prendre lorsqu'il est utilisé par un système de traitement : la cellule qui interprète la chaîne d'ADN, ou un processeur modifiant ses états internes en fonction de la lecture du code. Cela peut paraître décevant, mais c'est cependant d'une réalité confondante : il a été établi que, de part la différentiation cellulaire des organismes évolués, seule une partie du code génétique est interprété ("compris") par un groupe de cellule. De même, peu d'êtres humains ont développé le référentiel d'interprétation nécessaire pour "comprendre" les théories de la mécanique

[1] Ces aspects seront développés ultérieurement dansce document.

quantique, ou, plus difficile, de remplir sans erreur une déclaration d'impôt, ou encore de plier et déplier une carte routière sans la détériorer ([54]). Un système d'interprétation *comprend* un code lorsqu'il est capable de le traduire en actions sur son environnement interne ou externe. Comprendre, c'est transformer de l'information en connaissance opérationnelle. Définir la sémantique d'un code nécessite donc de tenir compte du ou des systèmes d'interprétation.

Dans le traitement des langages, la sémantique *statique* définit les propriétés du langage indépendamment de son exécution par un système d'interprétation [1]. Ce sont par exemple des règles de portée dans la déclaration des variables, des contraintes telle que l'unicité d'un nom de variable dans un contexte d'utilisation, mais aussi le contrôle des types ou des transformations (optimisations ou autre). J. Heering [59] utilise ASF+SDF, un formalisme générique de définition de langages pour spécifier la sémantique statique de *Pico*, un langage d'étude identique à *While* mais permettant la déclaration de variables typées. ASF+SDF (*Algebraic Specification Formalism + Syntax Definition Formalism* [60, 120, 121]) permet de spécifier de manière générique et modulaire le niveau lexical, syntaxique et dans une certaine mesure, sémantique, des langages textuels. Les définitions sémantiques utilisent des équations algébriques qui sont généralement implantées au moyen de règles de réécriture. Les définitions proposées pour *Pico* sont fort peu convaincantes quant à leur application en grandeur réelle, et utilisent des spécifications du second ordre, dont les problèmes d'implantation restent à explorer.

La sémantique *dynamique* concerne les processus temporels associés au langage tels que les successions d'actions déclenchées par une instruction ou les opérations sur les données. Trois grandes approches sont développées : la sémantique *opérationnelle*, *dénotationnelle* et *axiomatique* [104]. Des possibilités de conversions entre les différentes approches ont étés explorées avec un certain succès ([3]).

Sémantique opérationnelle.

La signification du langage est exactement décrite par la succession d'étapes de calculs intermédiaires. A chaque étape est associé l'état de la machine abstraite qui réalise le calcul. Ainsi chaque transition est elle-même définie par la transformation qu'elle opère sur l'état. Si ces transitions sont simples, il suffit d'utiliser une sémantique à "petits pas" (*small step semantics*), où une règle de réécriture suffit à exprimer la modification de l'état. Voici par exemple la sémantique des opérations arithmétiques du langage *While*, dont la syntaxe à déjà été présentée :

$$\begin{array}{lcll} n_1\ \text{'+'}\ n_2 & \Rightarrow & n_1 + n_2 & (r_1) \\ n_1\ \text{'-'}\ n_2 & \Rightarrow & n_1 - n_2 & (r_2) \\ n_1\ \text{'*'}\ n_2 & \Rightarrow & n_1 \times n_2 & (r_3) \end{array}$$

Dans le cas ou la transition est conditionnée de manière complexe, un outil plus puissant est nécessaire : la sémantique opérationnelle structurelle (*SOS, Structured Operational Semantics*) développée par Plotkin [91] puis très largement utilisée par la suite. Par exemple, les trois règles précédentes ne permettent pas de spécifier sans ambiguité le calcul de $1\ \text{'+'}\ 2\ \text{'*'}\ 3$. Leur application donne soit

$$1\ \text{'+'}\ 2\ \text{'*'}\ 3 \quad \overset{(r_1)}{\Rightarrow} \quad 3\ \text{'*'}\ 3 \quad \overset{(r_3)}{\Rightarrow} \quad 9$$

soit, en inversant l'ordre d'application des règles

$$1\ \text{'+'}\ 2\ \text{'*'}\ 3 \quad \overset{(r_3)}{\Rightarrow} \quad 1\ \text{'+'}\ 6 \quad \overset{(r_1)}{\Rightarrow} \quad 7$$

En sémantique opérationnelle structurée, des règles supplémentaires, de la forme

$$\frac{A_1 \Rightarrow A_1'}{A_1\ \text{'+'}\ A_2 \Rightarrow A_1'\ \text{'+'}\ A_2}\ (r_1 b) \qquad \frac{A_1 \Rightarrow A_1'}{A_1\ \text{'-'}\ A_2 \Rightarrow A_1'\ \text{'-'}\ A_2}\ (r_2 b)$$

$$\frac{A_1 \Rightarrow A_1'}{A_1\ \text{'*'}\ A_2 \Rightarrow A_1'\ \text{'*'}\ A_2}\ (r_3 b)$$

[1] Plus exactement, les concepts proposés à ce niveau d'abstraction ne sont pas directement reliés à un modèle de l'exécution

permettent de forcer l'associativité à gauche pour les trois opérateurs, et de lever l'ambiguité sémantique. Dans les règles, A_i représente une sous-expression quelconque, telle que définie par la grammaire de *While*. La règle $(r_1 b)$ par exemple signifie "pour calculer A_1 '+' A_2, il est nécessaire de calculer d'abord la sous-expression A_1". Le calcul précédent s'écrit alors sous forme d'un arbre de preuve :

$$\dfrac{\dfrac{1\ \text{'+'}\ 2 \Rightarrow 3\ (r_1)}{(1\ \text{'+'}\ 2)\ \text{'*'}\ 3 \Rightarrow 3\ \text{'*'}\ 3}\,(r_3 b) \quad \dfrac{}{(3\ \text{'*'}\ 3) \Rightarrow 9}\,(r_3)}{(1\ \text{'+'}\ 2)\ \text{'*'}\ 3 \Rightarrow 9}$$

Pour les lecteurs qui ne seraient pas familiarisés avec la puissance des langages, ces règles associées à la grammaire s'appliquent parfaitement à toute expression arithmétique quelle que soit sa complexité : l'expression $1 + 50 * 20 - 3 + 340 - 34 * 3 + 12 - 567 * 56 + 50 * 20 - 3 + 340 - 34 * 3 + 12$ possède une sémantique parfaitement définie, qui correspond à la fois à un résultat et à la succession d'étapes pour y parvenir.

Sémantique dénotationnelle.

Avec l'approche *dénotationnelle*, on ne s'intéresse pas aux étapes des calculs, mais uniquement à leurs résultats. C'est une approche mathématique, et non comportementale. Plus précisément, la sémantique est définie au moyen de fonctions s'appliquant sur des structures mathématiques définies à partir de la syntaxe [1]. La fonction possède une signature $\varepsilon : A \rightarrow entier$ définissant la structure acceptée en entrée (une expression de la grammaire) et la structure obtenue en sortie (entiers naturels). Conventionnellement, ε est définie par une série d'équations :

$$\begin{array}{l}
\varepsilon : A \rightarrow entier \\
\varepsilon[\![n]\!] = n \\
\varepsilon[\![A_1 + A_2]\!] = plus(\varepsilon[\![A_1]\!], \varepsilon[\![A_2]\!]) \\
\varepsilon[\![A_1 - A_2]\!] = plus(\varepsilon[\![A_1]\!], neg(\varepsilon[\![A_2]\!])) \\
\varepsilon[\![A_1 * A_2]\!] = mul(\varepsilon[\![A_1]\!], \varepsilon[\![A_2]\!])
\end{array}$$

L'expression précédente signifie alors :

$$\varepsilon[\![1\ \text{'+'}\ 2\ \text{'*'}\ 3]\!] = mul(\varepsilon[\![1\ \text{'+'}\ 2]\!], \varepsilon[\![3]\!]) = mul(plus(\varepsilon[\![1]\!], \varepsilon[\![2]\!]), 3) = mul(plus(1, 2), 3) = 9$$

Sémantique axiomatique.

Elle s'intéresse d'avantage aux propriétés des programmes, plutôt qu'à leur signification. Les outils théoriques sont la logique et les systèmes formels ou axiomatiques. Par exemple, pour raisonner sur les expressions arithmétiques de *While* afin établir leur parité, le système suivant, composé d'un axiome et d'un théorème constitue la base de ce raisonnement et définit la parité des entiers naturels.

$$\begin{array}{rcll}
\mathrm{pair}(n) & \Leftrightarrow & n \bmod 2 = 0 & (a_1) \\
\mathrm{impair}(n) & \Leftrightarrow & \neg\ \mathrm{pair}(n) & (t_1)
\end{array}$$

Une relation de *satisfaction* $\models$ est alors définie de la manière suivante :

$$\begin{array}{lcl}
A_1 + A_2 \models \mathrm{pair}(A_1 + A_2) & \Leftrightarrow & (\mathrm{pair}(A_1) \wedge \mathrm{pair}(A_2)) \vee (\mathrm{impair}(A_1) \wedge \mathrm{impair}(A_2)) \\
A_1 + A_2 \models \mathrm{impair}(A_1 + A_2) & \Leftrightarrow & (\mathrm{pair}(A_1) \wedge \mathrm{impair}(A_2)) \vee (\mathrm{impair}(A_1) \wedge \mathrm{pair}(A_2))
\end{array}$$

Ce système s'applique par exemple à l'expression $(1 + 2) + 3$, et on peut prouver qu'elle est paire de la manière suivante (des étapes ont été volontairement omises) :

[1] Les traitements sémantiques sont généralement définis sur une syntaxe abstraite, c'est à dire simplifiée par élimination des détails syntaxiques (mots clefs, séparateurs,...). Celle-ci se décrit également au moyen d'une grammaire hors-contexte, et la transposition de l'arbre syntaxique concret en arbre abstrait est simple

$$
\begin{array}{rcl}
(1+2)+3 \models \mathrm{pair}((1+2)+3) & \Leftrightarrow & (\mathrm{pair}(1+2) \wedge \mathrm{pair}(3)) \vee (\mathrm{impair}(1+2) \wedge \mathrm{impair}(3)) \\
& \Leftrightarrow & \mathrm{impair}(1+2) \wedge \mathrm{impair}(3) \\
& \Leftrightarrow & \mathrm{impair}(1+2) \wedge (\neg(3 \bmod 2 = 0)) \\
& \Leftrightarrow & \mathrm{impair}(1+2) \wedge (\neg(faux)) \\
& \Leftrightarrow & \mathrm{impair}(1+2) \\
& \Leftrightarrow & (\mathrm{impair}(1) \wedge \mathrm{pair}(2)) \vee (\mathrm{pair}(1) \wedge \mathrm{impair}(2)) \\
& \Leftrightarrow & \mathrm{impair}(1) \wedge \mathrm{pair}(2) \\
& \Leftrightarrow & vrai \wedge vrai \\
& \Leftrightarrow & vrai
\end{array}
$$

Cette introduction aux différentes formalisations de la sémantique est volontairement simplificatrice, ne prenant pas en compte les structures d'exécution, c'est à dire, la mémoire, les registres, piles et autres éléments qui constituent l'architecture de la machine interprétant le code [1]. Les mécanismes affectant les structures d'exécution doivent être pris en compte, directement dans les sémantiques opérationnelles, et indirectement sous forme de modèles mathématiques dans la sémantique dénotationelle. Ainsi la règle de grammaire $C \longrightarrow x$ ' :=' A exprime une sémantique d'affectation, c'est à dire trois opérations successives : trouver l'adresse mémoire de la variable (ou créer une place en mémoire et mémoriser son adresse), évaluer l'expression A, puis stocker le résultat à l'adresse requise. Si le système de type est dynamique, il faut éventuellement vérifier la concordance des types ou des opérations de coercition entre la phase 2 et 3. La mémoire se modélise, par exemple, par une structure $M = a_0 : v_0, \ldots, a_n : v_n$ ou les a_i, v_i représentent des couples *adresse*, *valeur*, et s'utilise au moyen d'une opération de stockage *store* qui accepte une adresse et une valeur en entrée et retourne une nouvelle mémoire dans laquelle la valeur a été ajoutée $\mathrm{store}(M, a_i, v_i) \longrightarrow M'$. Le mécanisme de résolution d'adresse se modélise par une table de symbole $T = n_i : a_i$ qui pour chaque nom de variable (unique) n_i associe une adresse mémoire (unique) a_i, et s'utilise via une fonction *find* qui prend un nom en entrée et retourne une adresse $\mathrm{find}(T, n_i) \longrightarrow a_i$. Ces définitions peuvent sembler très abstraites, cependant, elles sont réellement très proches des mécanismes bas-niveau des processeurs. En sémantique dénotationelle, l'opération C ' :=' A est définie par

$$
\begin{array}{l}
\varepsilon : C \to ((M \times T) \to M) \\
\varepsilon[\![x := A]\!](M,T) = store(M, find(T, n_x), \varepsilon[\![A]\!](M,T))
\end{array}
$$

En sémantique opérationnelle, la réécriture doit porter aussi bien sur le code (x ' :=' A) que sur les structures. Il est nécessaire d'utiliser une structure mémoire supplémentaire P, une pile, sur laquelle des opérations de plus bas niveau sont définies (*Push, Pop*), et autorisant les échanges de paramètres. Ainsi, la signature du système de réécriture, c'est à dire sa forme générale est $\langle M, T, P, C\rangle \Longrightarrow \langle M', T', P', C'\rangle$.

L'opération d'affectation est exprimée par le système suivant :

$$
\begin{array}{rcl}
\langle M,\ T,\ P,\ x := A; C\rangle & \overset{\text{aff}}{\Longrightarrow} & \langle M,\ T,\ P,\ A; Push(n_x); find; store; C\rangle \\
\langle M,\ T,\ P,\ Push(v); C\rangle & \overset{\text{push}}{\Longrightarrow} & \langle M,\ T,\ P + [v],\ C\rangle \\
\langle M,\ T \cup \{n_i : a_i\},\ P + [n_i],\ find; C\rangle & \overset{\text{find}}{\Longrightarrow} & \langle M,\ T \cup \{n_i : a_i\},\ P + [a_i],\ C\rangle \\
\langle M \cup \{a_i : v\},\ T,\ P + [v_i, a_i],\ store; C\rangle & \overset{\text{store}}{\Longrightarrow} & \langle M \cup \{a_i : v_i\},\ T,\ P,\ C\rangle
\end{array}
$$

Les règles décrivant les opérations arithmétiques s'écrivent

$$
\begin{array}{rcl}
\langle M,\ T,\ P,\ A_1 + A_2; C\rangle & \overset{+}{\Longrightarrow} & \langle M,\ T,\ P,\ A_1; A_2; add; C\rangle \\
\langle M,\ T,\ P,\ A_1 - A_2; C\rangle & \overset{-}{\Longrightarrow} & \langle M,\ T,\ P,\ A_1; A_2; sub; C\rangle \\
\langle M,\ T,\ P,\ A_1 * A_2; C\rangle & \overset{*}{\Longrightarrow} & \langle M,\ T,\ P,\ A_1; A_2; mul; C\rangle
\end{array}
$$

les opérateurs de base add, sub et mul travaillent sur la pile :

$$
\begin{array}{rcl}
\langle M,\ T,\ P + [v_1, v_2],\ add; C\rangle & \overset{\text{add}}{\Longrightarrow} & \langle M,\ T,\ P + [(v_1 + v_2)],\ C\rangle \\
\langle M,\ T,\ P + [v_1, v_2],\ sub; C\rangle & \overset{\text{sub}}{\Longrightarrow} & \langle M,\ T,\ P + [(v_1 - v_2)],\ C\rangle \\
\langle M,\ T,\ P + [v_1, v_2],\ mul; C\rangle & \overset{\text{mul}}{\Longrightarrow} & \langle M,\ T,\ P + [(v_1 \times v_2)],\ C\rangle
\end{array}
$$

[1] Ces structures sont de plus haut niveau lorsque les machines sont plus abstraites

De nombreux langages en grandeur réelle sont aujourd'hui formalisés par des sémantiques opérationnelles, favorisant des extensions, même complexes. *Eiffel //* est une version parallèle du langage *Eiffel*, dont la sémantique formelle est totalement décrite en [10] au moyen d'un modèle mémoire comportant des structures très riches.

Grammaires attribuées

Ce formalisme a été, une fois encore, proposé par Donald Knuth, pour prolonger naturellement le traitement syntaxique à base de grammaires hors-contextes. L'arbre syntaxique construit par l'analyseur reflète la structure des règles de la grammaire. De plus, chaque noeud de l'arbre contient des attributs qui globalement représentent la sémantique qui n'a pas pu être encodée dans la topologie de l'arbre. Les grammaires attribuées exploitent ces informations en associant des opérations de traitement d'attributs aux règles de grammaire, sous la forme de fonctions qui calculent la valeur d'un attribut en fonction d'autres attributs présents dans les noeuds de la production. Par exemple, la règle qui définit la syntaxe de l'addition dans *While*, peut s'écrire

$$A_0 \quad \longrightarrow \quad A_1 \text{ '+' } A_2 \quad [A_0.type = f(A_1.type, A_2.type)]$$

où f est une fonction calculant le type de la valeur résultat d'une addition en fonction du type des opérandes. La notation $A_0.type$ signifie "l'attribut de nom *type* associé au nœud A_0". Au niveau d'une production, on distingue les attributs *hérités*, dont la valeur est déja calculée par ailleurs, et les attributs *synthétisés* dont la valeur doit être calculée par l'équation sémantique associée à la règle. Les choses se compliquent lorsque l'on veut calculer l'ensemble des attributs d'un arbre syntaxique car il faut prendre en compte les inter-dépendances entre attributs. Des cycles peuvent apparaître, qu'il faut reconnaître et signaler comme erronés. Dans les approches les plus élaborées, un évaluateur est calculé après analyse de l'arbre et de ses dépendances, et optimisé pour traiter la grammaire. Certaines approches réalisent cette opération "au vol", dynamiquement, après chaque analyse syntaxique. Les points forts des grammaires attribuées résident dans leur efficacité pour traiter certains aspects des traitements sémantiques, et leur aptitude aux traitements incrémentaux. Leurs points faibles sont la pauvreté de l'expressivité, et la dépendance au schéma d'exécution de l'évaluateur. Carle et Pollock proposent une étude générale sur l'optimalité de l'évaluation incrémentale pour des grammaires attribuées dites *hiérarchiques* [25], c'est à dire spécifiables de manière modulaire. Ils différencient la phase de détermination des noeuds modifiés de la phase de réévaluation.

Systèmes de Réécriture

La réécriture est un puissant moyen de spécifier des transformations d'une manière déclarative, à l'aide d'un ensemble de règles. Les liens entre réécriture et grammaires, ainsi qu'entre réécriture et sémantique opérationnelle ont été évoqués précédemment et de nombreux autres liens avec des domaines théoriques variés sont établis. Cette approche bénéficie donc d'une large assise théorique dont on peut trouver une introduction détaillée en [44].

Généralités Un système de réécriture transforme une structure d'entrée en une succession d'étapes, jusqu'à la *terminaison* du processus. Il peut également continuer indéfiniment, en fonction soit des propriétés du système, soit de la structure d'entrée.

Termes. Les structures considérées sont très précisément définies à partir d' une construction mathématique, et sont appellées *termes*. Ceux-ci sont stratifiés : un terme d'ordre 0 est une constante appartenant à un "vocabulaire" $\mathcal{F}_0$; un terme du premier ordre est une fonction dont les paramètres d'entrée et

de sortie sont soit des éléments de $\mathcal{F}_0$, soit des variables. Ils appartiennent à un vocabulaire $\mathcal{F}_1$; un terme du second-ordre est une fonction de termes de $\mathcal{F}_0$ ou de $\mathcal{F}_1$, ou de variables sur ces termes. De plus, on définit une *arité* : les termes d'arité 0 sont des constantes, 1 des variables, $n \geq 2$ sont des fonctions admettant $n - 1$ paramètres. Pour le système de réécriture de *While*, représentant partiellement sa sémantique opérationnelle, les vocabulaires sont :

$$\begin{array}{rcc} \mathcal{F}_0 & = & \{add, sub, mul, find, store\} \\ \mathcal{F}_1 & = & \{Push\} \\ \mathcal{F}_2 & = & \{;\} \end{array}$$

Les vocabulaires seuls n'amèneraient pas la puissance expressive nécessaire. On leur adjoint des variables pouvant référencer des termes quelconques. Lorsqu'un terme ne comporte pas de variables *libres*, c'est à dire non définies, on dit qu'il est *fermé*. Dans le domaine de la réécriture, un terme t tel que $t = Push(add(x, sub(y, z)))$ est appréhendé comme une structure, et plus précicément, un arbre. Dans un sens logique, mais aussi topologique, $sub(y, z)$ est un sous-terme de t et sa position peut être précisément exprimée au moyen d'un code numérique. Ce dernier est établi en numérotant les branches de l'arbre : ici, la position de y est 1.2, et on note $t|_{1.2} = sub(y, z)$.

Substitutions. Chaque étape de la réécriture est réalisée en appliquant une des règles qui constituent le système, chacune étant de la forme $l \to r$, qui signifie l se réécrit en r. Pour pouvoir s'appliquer à un contexte u (terme fermé), la partie gauche l doit pouvoir se substituer quelque part, à une position p, dans u. Une substitution est formellement une application des variables "libres" du terme l dans un contexte u, plus une bijection sur les termes constants dans les symboles de u [1]. Lorsqu'une telle substitution l^σ existe pour un contexte u, on le note $u[l^\sigma]_p$.

La sémantique de l'application de la règle au contexte peut alors se définir exactement comme $u[l^\sigma]_p \to u[r^\sigma]_p$ (*application contextuelle* de la règle). Un système de réécriture est donc défini par la structure de ses termes $\mathcal{T}$ (signature), et une relation binaire $\overrightarrow{R}$ sur $\mathcal{T} \times \mathcal{T}$, *fermée* quant à l'application contextuelle.

Cela signifie en d'autres mots que, quelques soient les termes s et t de $\mathcal{T}$, le contexte u et la position p, $s \to t \Longrightarrow u[s^\sigma]_p \to u[t^\sigma]_p$. Encore plus intuitivement cela signifie que l'application d'une règle ne produira jamais une structure incompatible avec le système : si aucune règle ne peut s'appliquer, ce ne peut être que par défaut d'une substitution valide, et non parce que le contexte est devenu "incompréhensible" [2].

Ces définitions peuvent paraître arides, mais elle sont d'une indispensable précision pour aborder les subtilités de la réécriture. Ce qui est étonnant, dans la réécriture, c'est le contraste entre la rigueur formelle du cadre, et la liberté des hypothèses à l'intérieur de ce cadre. Ainsi, opérationnellement, c'est à la base un système non déterministe (n'importe quelle règle applicable peut être choisie, et à une position quelconque).

Principales propriétés. Lorsqu'un système de réécriture termine toujours quel que soit le contexte d'application, on dit qu'il est *normalisant*. Si toutes les dérivations envisageables pour un contexte donné conduisent toujours à la même forme normale, il est dit *confluent*. Dans de nombreuses applications, il est indispensable d'établir la confluence, notamment dans la compilation, o un arbre syntaxique abstrait, vu comme un terme, est réécrit en code objet (on souhaite dans ce cas

[1] lorsqu'on ne considère pas les variables, c'est seulement un *matching*

[2] Ceci est vrai du point de vue structurel mais faux en général si par exemple on associe une relation de typage aux termes : un terme correct structurellement peut prendre un type incorrect au cour de la réécriture ; la conservation du type est donc une proprit oprationnelle importante

que plusieurs compilations distinctes du même programme produisent le même code !). Ce sont des propriétés généralement difficiles à établir, et les techniques de preuves "générales" utilisent des raisonnements sur la forme globale des règles, ou certaines propriétés de recouvrement des membres gauches et droits. Dans le cas des machines abstraites (comme celle de *While*), la terminaison n'est pas considérée, car volontairement dépendante du code (un programme spécifiant une boucle infinie est licite, du point de vue de son exécution).

Variantes. De nombreuses variantes ont été proposées et étudiées (cf [66]). La première consiste en un regroupement des règles dans des systèmes différents, appliqués séquentiellement. Cette approche permet souvent de faciliter la terminaison et la confluence. Une autre variante est d'affecter des priorités aux règles, et donc de restreindre l'indéterminisme. Dans la même catégorie, on peut appliquer une stratégie implicite pour choisir les positions des substitutions dans le contexte (*redex*), comme par exemple, la *right most, innermost* pour désigner le redex le plus à droite et le plus "emboîté". L'expressivité peut être améliorée en utilisant des règles conditionnelles, qui ne sont déclenchées que si le contexte vérifie un prédicat (optionnel) particulier. Plus récemment, le langage *ELAN* développé dans le projet *Protheo* [93] propose un modèle de réécriture très expressif, où les stratégies d'application des règles sont elles-mêmes dynamiquement définies par réécriture ([18]).

Machines abstraites Leur première utilisation à été d'établir une notion théorique de calculabilité des langages, par le mathématicien Alan Türing. Cela correspond au problème fondamental, qui consiste à définir les limites des possibilités de calcul d'un processeur, qui n'est autre qu'une machine dont les opérations de base, en nombre limité, sont "mécaniquement" définies. Le principal résultat concerne la décidabilité des problèmes calculatoires. Pour un problème donné, existe-il un, ou des algorithmes (succession définie de calculs), capable de le résoudre en un temps fini ? Il existe de fait toute une classe de problèmes indécidables, pour lesquels aucune solution algorithmique ne peut être trouvée ; tout du moins, par une machine de Türing universelle. Dans ce cas, seules des approches "heuristiques" peuvent être envisagées. L'interêt vient du fait qu'une machine de Türing reflète fidèlement l'expressivité des processeurs actuels. Donc, plus concrètement, un chercheur désirant résoudre algorithmiquement un problème difficile devrait d'abord vérifier que celui-ci est décidable, afin de s'éviter un travail désespéré, ou des résultats erronés ! Bien entendu, établir la décidablité n'est pas une chose triviale, et généralement, on établit un isomorphisme entre le problème considéré et un des nombreux problèmes répertoriés comme non-décidables. Outre cette application fondamentale (la calculabilité des langages), les machines abstraites sont utilisées pour modéliser formellement des processeurs, compilateurs et interpréteurs. Le langage *While* a été utilisé par John Hannan pour illustrer son étude concernant la génération automatique de compilateurs à partir d'une spécification de la sémantique opérationnelle du langage [57]. Dans cette approche, la spécification prend la forme d'un système de réécriture de termes (*SRT*), similaire à celui que nous avons présenté en exemple. Ce système est en fait un interpréteur pour le langage. Il est possible d'appliquer un traitement automatique, appelé *séparation de passes*, permettant de générer automatiquement un compilateur et un *exécuteur* pour ce langage. Dans l'interprétation, les vérifications sémantiques et les transformations nécessaires pour l'exécution sont réalisées au vol, au fur et à mesure du déroulement du programme. L'avantage réside dans la souplesse et la rapidité du cycle de développement, puisqu'il n'y a pas de phase de compilation. L'inconvénient provient de la perte de performance, mais surtout d'un problème de fiabilité. Rendre un programme interprété robuste et fiable nécessite de développer des jeux de tests très complets pour compenser l'absence de vérification statique. En effet, il faut exécuter toutes les branches du programme afin de les valider (au moins les propriétés

basiques examinées par les compilateurs performants : types, anomalies de flots,...).

La séparation de passes distingue le code et les données, lors du processus de réécriture. Le principe est d'isoler les règles qui transforment uniquement le code de celles qui transforment les données. Les premières, indépendantes du contexte d'exécution, sont applicables dans une phase de compilation. Le problème revient alors à bâtir deux systèmes de réécriture distincts : le premier réalise la compilation, en réécrivant le code source dans un code intermédiaire. Le second exécute ce code intermédiaire comme un processeur virtuel.

Cette approche possède de nombreux avantages : d'une part, la spécification de l'interpréteur peut être utilisée directement pour implémenter celui-ci, à la condition d'être capable d'exécuter des machines abstraites générales. Cette dernière hypothèse ne pose pas de problèmes théoriques, et des travaux tels que ceux de Kamperman et Walters [64, 65] réalisent une bonne synthèse des techniques connues, et montrent son applicabilité (un langage ad-hoc *ARM* est utilisé pour spécifier des machines abstraites, qui sont compilées en code C avec des performances réalistes). De nombreux langages utilisent des machines abstraites comme cible de génération (Pascal, de nombreux langages fonctionnels comme ML, CAML et même Prolog avec la *WAM* (Warren Abstract Machine, du nom de son concepteur), et plus récemment Python et Java. Ici, il s'agit non pas d'implanter une machine abstraite particulière, mais de les générer à partir de leurs spécifications. En phase d'étude du langage, il est intéressant de disposer d'un interpréteur pour bénéficier de sa souplesse. La technique de séparation de passe permet ensuite, "gratuitement", de bénéficier d'un compilateur et d'un processeur virtuel. La difficulté théorique dans l'approche de Hannan consiste à établir la confluence du système de réécriture genéré pour la compilation, et à démontrer l'équivalence comportementale d'un programme compilé avec sa version interprétée. Il est également important que le compilateur soit optimal, en ce sens qu'il réalise effectivement la compilation de tout code indépendant du contexte d'exécution. L'auteur démontre ces propriétés à l'aide d'une technique connue [44], en s'appuyant sur une fonction d'abstraction arithmétique caractérisant la complexité structurelle des termes. Cette fonction permet d'établir que chaque règle du compilateur réduit la complexité du terme, et donc que le *SRT* est normalisant. Intuitivement, cela est logique, puisque le compilateur "abaisse" la complexité structurelle (comme un analyseur l'augmente). Nous produisons en illustration les *SRT*s obtenus pour la partie du langage *While* considérée jusqu'alors. Le compilateur est :

$$\begin{array}{rcl}
x := A; C & \stackrel{\text{aff}}{\Longrightarrow} & A; Push(n_x); find; store; C \\
v_i; C & \stackrel{\text{value}}{\Longrightarrow} & Push(v_i); C \\
A_1 + A_2; C & \stackrel{+}{\Longrightarrow} & A_1; A_2; add; C \\
A_1 - A_2; C & \stackrel{-}{\Longrightarrow} & A_1; A_2; sub; C \\
A_1 * A_2; C & \stackrel{*}{\Longrightarrow} & A_1; A_2; mul; C
\end{array}$$

l'exécuteur est décrit par :

$$\begin{array}{rcl}
\langle M, \quad T, \quad P, \quad Push(m); C\rangle & \stackrel{\text{push}}{\Longrightarrow} & \langle M, \quad T, \quad P + [m], \quad C\rangle \\
\langle M, \quad T \cup \{n_i : a_i\}, \quad P + [n_i], \quad find; C\rangle & \stackrel{\text{find}}{\Longrightarrow} & \langle M, \quad T \cup \{n_i : a_i\}, \quad P + [a_i], \quad C\rangle \\
\langle M \cup \{a_i : v\}, \quad T, \quad P + [v_i, a_i], \quad store; C\rangle & \stackrel{\text{store}}{\Longrightarrow} & \langle M + \cup\{a_i : v_i\}, \quad T, \quad P, \quad C\rangle \\
\langle M \cup \{a_i : v_i\}, \quad T, \quad P + [a_i], \quad get; C\rangle & \stackrel{\text{get}}{\Longrightarrow} & \langle M \cup \{a_i : v_i\}, \quad T, \quad P + [v_i], \quad C\rangle \\
\langle M, \quad T, \quad P + [v_1, v_2], \quad add; C\rangle & \stackrel{\text{add}}{\Longrightarrow} & \langle M, \quad T, \quad P + [(v_1 + v_2)], \quad C\rangle \\
\langle M, \quad T, \quad P + [v_1, v_2], \quad sub; C\rangle & \stackrel{\text{sub}}{\Longrightarrow} & \langle M, \quad T, \quad P + [(v_1 - v_2)], \quad C\rangle \\
\langle M, \quad T, \quad P + [v_1, v_2], \quad mul; C\rangle & \stackrel{\text{mul}}{\Longrightarrow} & \langle M, \quad T, \quad P + [(v_1 \times v_2)], \quad C\rangle
\end{array}$$

Ainsi le programme

$$a := 0; b := 5; a := b + (3 * b);$$

se compile en

$$\begin{array}{rcl}
C & = & Push(0); Push(a); find; store; Push(5); Push(b); find; store; \\
 & & Push(3); Push(b); find; get; mul; Push(b); find; get; add; Push(a); find; store
\end{array}$$

Son exécution dans le contexte

$$M = \{\alpha_a : 0, \alpha_b : 0\} \quad T = \{a : \alpha_a, b : \alpha_b\} \quad P = [\,]$$

produit la dérivation suivante :

$$\begin{array}{rl}
 & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [\,], \quad Push(0); Push(a); C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [0], \quad Push(a); find; C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [0, a], \quad find; store; C \rangle \\
\stackrel{\text{find}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [0, \alpha_a], \quad store; Push(5); C \rangle \\
\stackrel{\text{store}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [\,], \quad Push(5); Push(b); C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [5], \quad Push(b); find; C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [5, b], \quad find; store; C \rangle \\
\stackrel{\text{find}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 0\}, \quad T, \quad [5, \alpha_b], \quad store; Push(3); C \rangle \\
\stackrel{\text{store}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [\,], \quad Push(3); Push(b); C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [3], \quad Push(b); find; C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [3, b], \quad find; get; C \rangle \\
\stackrel{\text{find}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [3, \alpha_b], \quad get; mul; C \rangle \\
\stackrel{\text{get}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [3, 5], \quad mul; Push(b); C \rangle \\
\stackrel{\text{mul}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [15], \quad Push(b); find; C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [15, b], \quad find; get; C \rangle \\
\stackrel{\text{find}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [15, \alpha_b], \quad get; add; C \rangle \\
\stackrel{\text{get}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [15, 5], \quad add; Push(a); C \rangle \\
\stackrel{\text{add}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [20], \quad Push(a); find; C \rangle \\
\stackrel{\text{push}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [20, a], \quad find; store \rangle \\
\stackrel{\text{find}}{\Longrightarrow} & \langle \{\alpha_a : 0, \alpha_b : 5\}, \quad T, \quad [20, \alpha_a], \quad store \rangle \\
\stackrel{\text{store}}{\Longrightarrow} & \langle \{\alpha_a : 20, \alpha_b : 5\}, \quad T, \quad [\,], \quad \rangle
\end{array}$$

On peut remarquer que la mémoire M contient le résultat du calcul, à l'adresse α_b. Les schémas d'affectation du type a **:=**$b + op(A, b)$, dans l'exemple, $op(A, b) = 3 * b$ peuvent être optimisés en modifiant les opérations mémoire. Le code normalement produit est

$$Push(3); Push(b); find; get; mul; Push(b); find; get; add$$

, où l'on constate deux séquences d'accès mémoire $Push(b); find; get$. Le code optimisé pourrait être

$$Push(3); Push(b); find; get; dup; mul; rot; add$$

, utilisant deux nouvelles instructions rot et dup, courantes dans les opérations de pile, dont la sémantique est définie par

$$\begin{array}{rcl}
\langle M, \quad T, \quad P + [v_i], \quad dup; C \rangle & \stackrel{\text{dup}}{\Longrightarrow} & \langle M, \quad T, \quad P + [v_i, v_i], \quad C \rangle \\
\langle M, \quad T, \quad P + [v_i, v_j], \quad rot; C \rangle & \stackrel{\text{rot1}}{\Longrightarrow} & \langle M, \quad T, \quad P + [v_j, v_i], \quad C \rangle \\
\langle M, \quad T, \quad P + [v_i, v_j, v_k], \quad rot; C \rangle & \stackrel{\text{rot2}}{\Longrightarrow} & \langle M, \quad T, \quad P + [v_j, v_k, v_i], \quad C \rangle
\end{array}$$

le compilateur pourrait alors comprendre une règle d'optimisation utilisant le fait que les instructions de pile sont plus rapides que les accès à la mémoire :

$$\begin{array}{rcl}
C_1; Push(x_n); find; A; Push(x_n); find; C_2 & \stackrel{\text{opt1}}{\Longrightarrow} & C_1; Push(x_n); find; dup; A; rot1; C_2 \\
C_1; Push(x_n); find; A; Push(x_n); find; C_2 & \stackrel{\text{opt2}}{\Longrightarrow} & C_1; Push(x_n); find; dup; A; rot2; C_2
\end{array}$$

Le choix de $opt1$ ou de $opt2$ depend du nombre d'opérandes entrées dans la pile par A. De telles règles peuvent s'appliquer plusieur fois, comme par exemple pour le code source b **:=** $b + 3 * b$, qui se compile, sans optimisation, en

$$C = Push(b); find; get; Push(3); Push(b); find; get; mul; add; Push(b); find; store$$

Après une première application de $opt2$, le code se réécrit en

$$\begin{cases} A & = & get; Push(3) \\ C_1 & = & \emptyset \\ C_2 & = & get; mul; add; Push(b); find; store \\ C & = & Push(b); find; dup; get; Push(3); rot2; get; mul; add; Push(b); find; store \end{cases}$$

puis ensuite, par $opt1$, en

$$\begin{cases} A & = & dup; get; Push(3); rot; get; mul; add; \\ C_1 & = & \emptyset \\ C_2 & = & store \\ C & = & Push(b); find; dup; dup; get; Push(3); rot2; get; mul; add; rot1; store \end{cases}$$

Le lecteur peut s'assurer de l'équivalence du code normal et optimisé en appliquant le *SRT* de l'exécuteur à ces deux séquences de code. Une règle de cette puissance ne peut toutefois pas être traitée directement dans un *SRT*, car il faut d'une part vérifier les opérations de pile réalisées par la substitution A (choix de $opt1$ ou $opt2$), et d'autre part, prendre en compte les problèmes d'associativité car le terme gauche et ses trois variables ne peut pas être décrit par un arbre de manière unique. En effet, la notation $A; B$ signifie en fait $succ(A, B)$, où $succ$ est le nom logique du constructeur de séquences ;. Une expression de la forme $C_1; V; C_2$ ou V est une variable ne peut donc être exprimée que par un arbre de la forme $succ(C_1, succ(V, C_2))$, et l'unification de $C_1; V; C_2$ avec, par exemple, $C_1; a; b; C2$, s'écrivant $succ(C_1, succ(a, succ(b, C_2)))$ doit utiliser des étapes intermédiaires, telles que décrites par

$$succ(X, succ(Y, succ(Z, W))) \rightarrow succ(X, succ(succ(Y, Z), W))$$

Ces règles sont généralement délicates à utiliser car elles conduisent rapidement à des transformations sans terminaison. Cette illustration a pour but de montrer comment les contraintes liées aux structures affectent profondément les traitements.

3.3 Outils de génération

Cette section présente des outils qui utilisent des spécifications de haut-niveau pour générer des composants de transformation applicables au traitement des langages.

3.3.1 Progress : un langage et environnement de spécification par réécriture de graphes

Progress [105], est un langage de très haut niveau qui réalise une synthèse de différents horizons : grammaires de graphes, bases de données, langages visuels diagramatiques (basés sur des diagrammes, graphiques de type "entité-relation") et logique. *Progress* génère des composants *Modula-2*, *C* ou *C++* pouvant être intégrés à des applications générales. Il est particulierement bien adapté au traitement des problèmes pouvant être modèlisés par des graphes, et dont les temps de résolution ne sont pas fortement déterministes.

modèle de graphe

Progress traite des graphes dont les noeuds sont typés et attribués et dont les arcs, orientés, sont également typés. Le typage en question intègre la notion d'héritage, dans les deux cas. De plus, les définitions des types de noeuds et d'arcs autorisent un polymorphisme paramétrique explicite. Une distinction est faite entre types abstraits (nommés *class*) qui décrivent l'interface des noeuds et leurs implémentations (nommées *type*). L'héritage porte sur les types abstraits. On peut donc affirmer que les graphes de Progres sont globalement décrits par le type des noeuds et les types des liens. Ces derniers

peuvent de plus spécifier des contraintes de cardinalité dans leurs interconnexions. Ainsi, les propriétés topologiques des graphes ne sont abordées que *localement* (Courcelle [39] présente les problèmes d'expressivité liés aux propritétés topologiques des graphes généraux). Les graphes sont physiquement manipulés au moyen d'une base de données spécialisée *GRAS* , dont les opérations transactionnelles portent sur les noeuds et les arcs.

modèle de réécriture

Dans la définition théorique du langage, les productions sont formalisées par des prédicats logiques, assertions sur la structure du graphe. Une production peut décrire une partie gauche complexe pouvant désigner des ensembles de noeuds non déterminés en cardinalité. Les sous-graphes filtrés lors de l'application de la production peuvent toutefois être réécrits, les ensembles étant alors manipulés en blocs dans la partie droite. Les arcs pendants ou cachés sont systématiquement détruits afin d'assurer la cohérence de la réécriture. Des conditions logiques supplémentaires portant sur les types ou relations des noeuds sont spécifiables en partie gauche (ces conditions utilisent les connectives logiques standard sur des prédicats de types et de relations). Par ailleurs, lorsqu'une production est appliquée, les attributs des noeuds sont réévaluables au moyen d'appels de fonctions explicites.

contrôle de la réécriture

Un des aspects original de *Progress* est d'associer des possibilitées de contrôle [124] dans l'application des productions (ces dernières sont explicitement nommées, donc manipulables par désignation). Essentiellement, l'application des règles est régie par des transactions, qui sont emboîtables, et utilisent elles-mêmes des primitives d'itération et de composition de règles. L'opérateur *or* (P_1 or $P2$ or ...) choisit une production P_i applicable, sans déterminisme. L'opérateur *and* (P_1 and P_2 and ...) demande l'application de toute les productions P_i, dans un ordre quelconque. L'opérateur & (P_1 & P_2 & ...) demande l'application de toutes les productions P_i, dans l'ordre spécifié. Des sélections conditionnelles multiples (*choose*) et les instructions *loop* et *skip* complètent les moyens de contrôles mis à disposition du programmeur. L'exécution d'une spécification *Progress* développe un arbre de recherche à la Prolog, et effectue des retours arrières lorsque une transaction ne peut être validée.

Environnement

L'environnement de développement est très complet pour un prototype de recherche, et comporte un éditeur textuel/visuel mixte, couplé à un analyseur syntaxique incrémental, un interpréteur et metteur au point intéractif et un outil externe de visualisation de graphes. L'édition visuelle n'est pas indispensable, car la transposition textuelle/visuelle est complète. Toutefois, elle se révèle indispensable pour maîtriser l'expressivité des productions.

Génération

L'utilisateur peut lancer un processus de génération de code source C ou de code source Modula-2. Le code compilé offre un facteur d'accélération d'environ 10 (cf. [106]), car les opérations liées à la base de données ne sont pas compressibles. Les modules générés sont intégrables dans des applications externes, associés à des librairies spécifiques pour exploiter la base de données. L'intégration des mécanismes de retour arrière ne semble pas très directe.

Synthèse

Progres se base sur une sémantique statique et dynamique précise et formalisée, un environnement de développement évolué, une forte expressivité et des moyen de contrôle de l'exécution riches et variés. Toutefois, les mécanismes de retour arrière sont difficile à maîtriser, et le couplage fort à une base de donnée peut se révéler comme un obstacle à l'intégration et à la performance de l'exécution.

L'association de mécanismes de sous-typage et de polymorphisme se montre particulièrement fertile, favorisant la simplicité et la réutilisation des spécifications. Le travail est relativement réçent, peu connu, et donc son utilisation semble restreinte. Il a été utilisé, entre autres, pour prototyper des outils de modélisation de procédés (cf [107]).

3.3.2 Optimix : réécriture de graphes pour l'optimisation de code

Optimix [9, 8] est un langage spécialisé pour traiter l'optimisation de codes dans les compilateurs. Il s'appuie sur un formalisme bien développé et sur une implantation conséquente. La principale innovation d'Optimix est d'utiliser la réécriture de graphes pour réaliser les transformations complexes nécessaires aux processus d'optimisation.

Le modèle de graphe

Il prend en compte divers types de noeuds mais aussi divers types de liens. Les premiers ont pour vocation de modéliser les instructions du code intermédiaire ou autres informations provenant de l'analyse du programme source. Les seconds permettent d'exprimer les relations logiques entre les diverses entités considérées. On peut citer les relations de dépendance dans l'évaluation d'expressions, les relations décrivant l'écoulement du flot de contrôle. D'une manière générale, les graphes dans Optimix sont dirigés, relationnels et étiquetés. Formellement, si $\Sigma = \Sigma_N \cup \Sigma_A$ est un ensemble de constantes (étiquettes), Σ_N et Σ_A étant les ensembles de constantes étiquettant les noeuds et les arcs, un Σ-graphe $G = (N, A, netiq)$ est décrit par N, un ensemble de noeuds, $A \subseteq (N \times N \times \Sigma) = \{A_a \subseteq N \times N\}_{a \in \Sigma_A}$, un ensemble fini d'arcs étiquetés définis sur $N \times N$. Cette définition autorise des arcs multiples entre deux noeuds, à condition qu'ils aient des étiquettes distinctes. Le modèle utilise également une version "négative" des graphes, c'est à dire un $\Sigma^\neg$-graphe $J = (N, A, netiq, neg)$ muni d'une fonction booléene $neg : A \rightarrow \mathcal{B}$ qui définie les arcs ne *devant pas* être présents dans le redex. Le graphe J^{pos} est obtenu par $J^{pos} = (N, A - \{a \in A | neg(a) = vrai\}$. Les arcs négatifs sont utilisés dans la réécriture pour augmenter l'expressivité tout en conservant de l'efficacité.

Le modèle de réécriture

Optimix utilise une approche spécifique, orientée vers l'efficacité. Celle-ci est proche des techniques algébriques et leur condition d'adhérence, mais autorise les "arcs cachés" entre des noeuds devant être détruits, ainsi que les arcs pendants, automatiquement détruits. Comme le fait remarquer l'auteur, les règles ne sont donc plus reversibles : le système ne prétend pas faire de l'analyse syntaxique, mais uniquement de la transformation par réécriture.

Une règle Optimix s'écrit $(L \supseteq K \subseteq R)$, L membre gauche appartenant à $\mathcal{L}(\Sigma^\neg)$ l'ensemble des graphes négatifs, R membre droit appartenant à $\mathcal{L}(\Sigma)$, et K, invariant dans la réécriture, appellé graphe d'interface. L'application d'une règle r à un graphe G requiert trois conditions :

1. Filtrage. Un morphisme injectif de $g : L^{pos} \rightarrow G$, appellé carte d'occurrence, doit être établi. $g(L^{pos})$ est appellé *redex*. L'utilisation de L^{pos} indique que pour cette première condition, les arcs

négatifs ne sont pas considérés. Des arcs cachés peuvent exister dans le redex. Les arcs entre $G - g(L^{pos})$ et $g(L^{pos})$ peuvent exister (arcs pendants et cachés). Formellement, le morphisme n'établit pas un sous-graphe, mais un filtrage (*matching*). L'auteur justifie cette approche en expliquant qu'elle est plus souple quant à l'évolution de la structure du graphe : si une nouvelle relation est ajoutée, les règles définies auparavant restent applicables. Dans l'approche basée sur les sous-graphes, il faudrait les réécrire intégralement.

2. test des arcs négatifs. Un arc négatif etiqueté l, $a_l = (n_1, n_2)$ reliant deux noeuds n_1 et n_2 de L doit impliquer $\neg(\exists a'_l = (g(n_1), g(n_2)))$.
3. test d'addition d'arcs. L'application de la règle doit créer au moins un nouvel arc entre les noeuds invariants du redex. Cette condition permet de garantir la terminaison du processus de réécriture.

Lorsque ces conditions sont remplies, la production est appliquée au graphe G afin de créer un nouveau graphe H en cinq étapes successives :

1. destruction des arcs du redex
2. destruction des arcs cachés
3. destruction des noeuds
4. création des nouveaux noeuds
5. création des nouveaux arcs

Terminaison

L'auteur distingue plusieurs types de terminaisons : par accumulation d'arcs ou noeuds (les règles augmentent la complexité du graphe de manière monotone ; comme seul un arc d'étiquette donnée est pris en compte dans la condition de déclenchement d'une règle, la terminaison est assurée), par élimination d'arcs (inverse). Les systèmes de réécriture exhaustifs combinent les deux stratégies, et leur application conduit toujours à des formes normales.

Stratification

Les systèmes exhaustifs ne sont pas convergents : plusieurs graphes différents peuvent être obtenus à partir du même graphe d'entrée. L'auteur propose une approche heuristique basée sur la stratification des règles afin d'obtenir une forme normale unique. Le principe repose sur une analyse statique des dépendances entre règles, sur la base des opérations d'addition et de destruction. La stratification permet d'appliquer d'abord les règles qui expansent le graphe, puis celles qui le réduisent. L'ordre d'application au sein des strates est fixé par une relation partielle. Certains systèmes ne sont pas stratifiables, et ne peuvent donc pas converger.

Le langage

Il est modulaire et déclaratif, et le programmeur doit spécifier le type de réécriture, *addition d'arcs* ou *exhaustif*. Dans ce dernier cas, les opérations de créations et destructions de noeuds et d'arcs sont explicitées dans un ordre fixe. Les arcs sont représentés textuellements par des prédicats binaires. Les strates sont spécifiées explicitement par le développeur. Des appels C externes peuvent être déclenchés lors des tests des prédicats relationnels, à la condition que ces fonctions externes soit déclarées ainsi que leurs signatures.

Synthèse

Optimix, au travers d'une approche théorique rigoureuse et d'une implantation soignée, démontre que la réécriture de graphes est applicable à des problèmes réels et difficiles, tels que l'optimisation de programmes. L'auteur propose une stratégie de réécriture efficace dont il tire partie pour stratifier et ordonner automatiquement l'application des règles, à partir de l'analyse statique de leurs dépendances. Cette stratification et cet ordre, quand ils existent, permettent de garantir la convergence du système, c'est à dire sa terminaison sur une forme normale unique. Toutefois, les transformations complexes, mettant en oeuvre un nombre indéfini de noeuds, ne peuvent pas être spécifiées avec Optimix.

3.3.3 Centaure : environnements intégrés et sémantique naturelle

Centaure [19] est un générateur d'environnements intégrés utilisant des spécifications sémantiques de haut-niveau (*sémantique naturelle*), dans un langage appellé *Typol* [45]. Il se situe dans le prolongement du système *Mentor* et son langage associé *Metal* [79], qui travaillait au niveau de la syntaxe concrète et abstraite pour offrir des éditeurs syntaxiques, et décompilateurs génériques, c'est à dire utilisant une spécification de haut-niveau d'un langage quelconque.

Architecture

L'environnement *Centaure* se compose d'un noyau fonctionnel spécialisé, de langages de spécifications et d'un environnement interactif.

1. Noyau fonctionnel. Il se compose d'une machine abstraite de transformation d'arbre (appellée VTP, *Virtual Tree Processor*) et d'une machine de traitement symbolique. Le VTP, écrit dans un dialecte *Lisp*, offre des primitives pour transformer les arbres syntaxiques concrets et abstraits au travers d'une interface précisément définie. Le moteur logique est un interpréteur *Prolog*, également accédé au travers d'une interface. Les interfaces de communication entre les deux composants permettent de déclencher des évaluations Prolog depuis les fonctions du VTP, mais également, de commander des transformations en cours d'évaluation de prédicats Prolog.
2. Spécifications. Les syntaxes concrètes et abstraites sont spécifiées en *Métal*, et produisent des analyseurs syntaxiques. Des règles de grammaires sont annotées par des appels fonctionnels permettant de construire des arbres abstraits en cours d'analyse. Un langage spécialisé, *PPML*, permet de générer des décompilateurs, c'est à dire des outils permettant de regénérer des textes sources à partir de l'arbre abstrait. L'intêret est multiple : adapter les sources à différents utilisateurs, "normaliser" les conventions d'écriture, et même envisager des transports automatiques de programmes entre différents langages ou différentes versions. De plus, PPML est suffisamment puissant pour pouvoir générer des représentations de grande qualité graphique, au moyen d'un langage de description de boîtes similaire à LaTeX. Une spécification PPML se présente sous la forme d'un ensemble de règles dont la partie gauche est un terme de l'arbre abstrait, et la partie droite une chaîne d'instructions graphique, de texte, ou de variables permettant le traitement récursif de l'arbre. Les règles PPML sont vérifiées par un contrôleur de types qui s'assure que toutes les productions sont réellement utilisables (une règle peut être englobée par une autre, plus générale), et qu'elles sont conformes à la structure de l'arbre abstrait. Les transformations non spécifiées sont traitées par des règles de formatage "standard". Un programme PPML est traduit en Lisp. Les spécifications sémantiques sont exprimées en Typol, et sont décrites dans la sous-section suivante.
3. Environnement utilisateur.

Il est architecturé autour de trois idées fortes : des objets graphiques structurés, ayant un comportement explicitement spécifié en Esterel (langage de définition d'automates synchrones) ; des vues multiples sur les objets graphiques ; des menus contextuels et des boîtes de dialogue permettant de paramétrer le comportement du système. L'interface est bâtie au dessus de boîtes à outils Lisp, afin d'assurer une certaine portabilité, mais semble souffrir de ce fait d'une réduction de performance importante.

Spécifications en Typol

Typol est à la fois un formalisme et un langage complet, offrant une forte expressivité et une bonne modularité [45]. Une spécification typol est un ensemble de règles logiques, de complexité variable, allant du simple jugement, aux équations complexes utilisant des variables intermédiaires. De plus Typol est statiquement typé, bien que n'offrant pas la possibilité de construire de nouveaux types, et compilé. La vérification de types concerne l'utilisation d'arbres abstraits, l'importation de définitions externes, l'utilisation des variables et la définition d'opérateurs externes. Les variables sont de types *string*, *integer* ou *path*, ou des non-terminaux de la grammaire. Les opérateurs externes sont faiblement couplés au langage (pas d'encodage dans la génération), et se présentent comme des signatures offrant des facilités d'appel à des prédicats Prolog.

Les constructions du langage sont riches et nombreuses :

1. Opérateurs. Ils sont atomiques (feuilles des arbres abstraits), d'arité fixe (non-terminaux) ou des listes de taille dynamique. Des *patterns* permettent de filtrer des listes, c'est à dire des sous-arbres.
2. Propositions. Ce sont soit de simples prédicats (comme `IS-BOOLEAN(x)`), soit des relations quelconques (`x:T`, ou `p->c`) prises dans un vocabulaire prédéfini, et dont le sens est donné par leur utilisation dans le contexte, soit enfin des relations étiquetées, comme `s{i->o}r`, où les labels `i` et `o` sont des expressions Typol.
3. Sequents. Un séquent $H \vdash S$ est constitué d'une hypothèse H (liste de propositions) et d'une proposition S appellée *sujet*. La signification est "la proposition S doit être évaluée à partir des hypothèses H".
4. Règles d'inférence et axiomes. Elle sont constituées d'un dénominateur (séquent ou proposition unique, *sujet* de la règle) et d'un numérateur (liste de sequents ou propositions). Leur signification est "le dénominateur est vrai si tout les éléments du numérateur sont vrais". Les axiomes n'ont pas de numérateur.
5. Variables. Les variables locales à une règle indiquent que des mécanismes d'unification doivent être mis en œuvre pour évaluer la règle. Les variables globales, déclarées comme telles, sont typées et peuvent être référencées dans différentes règles ou propositions.
6. Règles Typol. Elles ont une structure particulière : un nom logique, une règle d'inférence et des informations sur les conditions d'évaluation ainsi que les actions, qui n'influencent pas le processus d'évaluation, mais peuvent utiliser les variables locales et globales. Ces actions sont déclenchées lorsque le numérateur de la règle a été évalué à vrai. Toutefois, il n'y a pas de contre-actions possibles lorsque le moteur prolog effectue une opération de retour arrière (*backtracking*).

De plus, des facilités sont proposées pour structurer les programmes Typol de manière modulaire :

1. Programmes. Ce sont les unités de compilation (nommées) comprenant une section de déclaration de variables globales et un corps de définition. Ce dernier comporte une section de définition de variables locales et une section pour les règles typol ou des ensembles de règles.

2. Ensembles. Un ensemble (*Set*) est une collection de règles, nommée et considérée comme un système formel indépendant, pouvant être invoqué au cours de l'évaluation d'une règle dans un autre système formel. Les ensembles sont importés au moyen d'une clause *Import* qui permet au passage de renommer leurs définitions internes.
3. Séquents nommés. C'est précisément le moyen de lancer l'évaluation d'un séquent à l'interieur d'un ensemble de règles données.

Synthèse

Centaure offre une grande richesse dans l'expression de la sémantique. De plus, le niveau d'abstraction de la sémantique naturelle correspond exactement aux concepts manipulés par les théoriciens des langages. Ce système a fait l'objet de nombreuses expérimentations qui ont démontré son applicabilité. Par exemple, le parallélisateur *ParaGraph* [46], s'appliquant à transformer un sous-langage de Fortran, est spécifié en Typol (sémantique statique et dynamique). Un système de transformation de documents [7] (formats LaTeXet *Tioga*) montre également la généralité de l'approche.

Toutefois, la mise au point reste très dépendante de Prolog, et le niveau de performance est probablement difficile à maîtriser. D'autre part, l'approche incrémentale semble difficile, bien que certains travaux établissent des ponts entre sémantique naturelle et grammaires attribuées ([11]). De plus, si la modularité est prise en compte au niveau du source, elle n'apparait pas au niveau de l'exécution. Des travaux plus récents proposent une architecture distribuée, plus ouverte, de type bus logiciels, ou environnements à diffusion de messages [31].

3.4 Conclusion

Ce chapitre propose une introduction aux principaux axes théoriques, infrastructures des traitements des langages. Le rôle des structures comme élément central des processus de transformation apparait comme un fil conducteur susceptible de nous guider vers de nouvelles évolutions. En particulier, elles sont le point de jonction entre les langages visuels et textuels. Dans la continuité des progrès apportés par les grammaires formelles et les expressions régulières pour la génération des analyseurs syntaxiques et lexicaux, la tendance est d'utiliser des spécifications de la sémantique des langages pour générer les divers composants nécessaires à leur traitement. La tâche est cependant beaucoup plus ardue du fait de l'étendue des domaines couverts par l'analyse sémantique. De plus, les langages ont évolué, entraînant des abstractions de données et d'exécution plus riches. En outre, ces derniers prennent en considération des facteurs externes susceptibles d'accroître la difficulté des traitements, tels que le parallélisme, la distribution, la mobilité, l'interopérabilité. Les traitements sémantiques considérés jusqu'à présent sont relativement différenciés dans leur étude, mais peu dans leur implantation. Leur génération, quant à elle, reste au stade du laboratoire et les chercheurs proposent soit des solutions viables très spécialisées, soit des solutions générales ne présentant pas les critères de modularité et d'efficacité requis pour répondre au syndrôme de Babel évoqué dans le chapitre précédent. Quelques uns des paradoxes à résoudre sont liés aux qualités du méta-langage recherché : *simple* pour décrire des processus complexes, *spécialisé* pour prendre en compte la spécificité des traitements langagiers, *général* pour couvrir de manière homogène la globalité des besoins.

Les graphes sont probablement les structures possédant la richesse et l'universalité attendues. Ils sont déjà utilisés pour modéliser un grand nombre de problèmes et d'algorithmes spécifiques à la compilation,

comme l'allocation de registres par coloration de graphes, l'analyse de flots de données, l'optimisation, le traitement des problèmes de précédence dans les évaluateurs. Toutefois, les outils de transformation de graphes ne présentent pas la puissance attendue, en termes d'expressivité et de souplesse. Les langages de spécification sont très complexes, ne parvenant pas à proposer une synthèse simplificatrice des structures de graphes. *Progress* utilise un modèle de données très (trop ?) riche permettant de générer des analyseurs aux prix de contraintes plus fortes sur les règles. Optimix, propose son propre modèle, adapté à l'optimisation de programmes, mais de ce fait très spécialisé.

Les machines abstraites, basées sur des systèmes de réécriture de termes sont d'un niveau suffisament bas pour être général, mais sont limitées dans leur application, principalement de part les contraintes structurelles imposées sur les termes. Ces contraintes ont pour but de rendre les spécifications exécutables et d'autoriser des transformations automatiques du système. De plus, leur utilisation concrète, sous forme d'un langage de programmation n'a pas suffisamment été explorée et expérimentée [1].

L'aspect architectural, dont nous avons évoqué au chapitre précédant l'importance pour l'intégration des outils, n'est pas pris en compte, probablement parce que les outils de génération basés sur la sémantique n'ont pas encore atteint la maturité requise. *Centaure* propose un pas dans cette direction, en associant un modèle d'architecture aux spécifications sémantiques : les spécifications ont vocation à s'intégrer dans un environnement de développement prédéfini. Toutefois, ce modèle est figé, en ce sens que les noyaux de traitements ne sont pas a priori réutilisables dans un autre contexte. *Centaure*, et les travaux associés (*Minotaure* [11]) représentent probablement l'avancée la plus importante dans le domaine de la génération d'outils pour le traitement des langages. Toutefois, des choix de base, tels que l'utilisation d'un formalisme de très haut-niveau (*Typol* [45]) pour spécifier la sémantique recèlent des inconvénients importants (si l'on considère le cadre de notre reflexion), tels que l'utilisation d'un moteur Prolog pour évaluer les règles, fortement monolithique, offrant peu d'ouverture, et très peu d'aptitudes aux traitements incrémentaux [11]. De même, la modularité, dans cette approche, n'a pas été considérée comme un point central, et surtout, la structure la plus évoluée reste l'arbre abstrait. L'intégration, au niveau des spécifications, est partiellement réalisée, puisque les décompilateurs sont spécifiés dans un langage basé sur le *pattern matching*, très différent de Typol. De plus, s'il est raisonnable de considérer qu'un concepteur de langage et outils spécialisé dispose de connaissances "pointues" dans ce domaine, la manipulation de la sémantique naturelle demande une compétence particulière, surtout s'il est nécessaire de mettre au point les règles en se basant sur l'interprétation et plus particulièrement le backtracking Prolog [45].

Pour conclure, dans les perspectives que nous avons esquissé, il semble que les différentes pièces du puzzle soient présentes, et qu'il reste à les assembler afin de proposer un méta-langage de plus bas niveau que *Centaure*, mais plus universel et plus intuitif, manipulant des structures de données riches, au moyen de machines abstraites généralisées. Ce langage possèderait un système de type polymorphe, des mécanismes de sous-typage et de modularisation puissants pour maximiser la réutilisation et minimiser les coûts d'évolution. Des fonctions de composition évoluées pourraient permettre de combiner des machines abstraites avec une grande richesse, en tirant partie de la simplicité des hypothèses d'exécution (on pense par exemple à l'inverse de l'opération de séparation de passes : spécifier une sémantique sur une machine abstraite déjà décrite par ailleurs). De plus, ce méta-langage pourrait être implanté sur une plateforme *Python* ou *Java* et *C/C++*, afin de bénéficier d'une réelle portabilité, et une réelle ouverture sur le monde industriel. Outre la prise en compte des traitements classiques (optimisation, contrôle des types, vérifications sémantiques), il pourrait s'appliquer à des outils visuels interactifs fortement couplés

[1] Les travaux de [65] vont dans ce sens, mais sous forme très basique, n'offrant pas les opérations et les primitives nécessaires à un puissant langage de transformation. De plus, le langage proposé dissimule la structure fondamentale de la réécriture, simple et directe, sous une abstraction plutôt obscure

aux noyaux de traitements sémantiques, et ainsi ouvrir la voie à une nouvelle génération d'outils.

A l'issue de cette reflexion sur l'état de l'art, les grands axes de notre méta-langage peuvent donc être établis de la manière suivante :

niveau d'abstraction. Afin de généraliser les applications potentielles du méta-langage, nous proposons deux niveaux d'abstraction complémentaires. Le premier, dit *intermédiaire*, permettra de définir les *structures* (les données et leurs types) ainsi que les machines abstraites réalisant leur transformation. Il sera de plus haut niveau que des langages généralistes tels que *C* ou *Java*, mais de plus bas niveau que les descriptions de grammaires formelles ou les équations sémantiques à la *Centaure*. Le deuxième, dit de *haut-niveau*, sera adapté à la spécification de grammaires et pourra être ultérieurement étendu à d'autres formalismes similaires. La génération d'analyseurs lexicaux et syntaxiques construira des structures de données et machines abstraites de niveau intermédiaire. Le concepteur pourra alors réutiliser ces composants en les intégrant dans ses propres spécifications, en bénéficiant de la spécialisation du niveau intermédiaire.

abstractions d'exécution. Elle devront permettre un contrôle *gradué* du déterminisme, afin de maîtriser les performances d'exécution, mais aussi d'autoriser une composition plus fine des composants de traitement qui seront générés. De plus, pour répondre aux besoins induits par l'interactivité des langages visuels, mais aussi pour augmenter l'expressivité des transformation, des opérateurs de composition *à l'éxécution* devront permettre l'exécution parallèle et séquentielle des transformations.

abstractions de communication. Elle devront faciliter l'échange d'information entre les composants de traitement, tout en préservant leur isolation par rapport au contexte, condition *sine qua non* d'une réutilisation effective. Elles pourront s'inspirer des modèles de coordination généraux, tels que *Linda* [50, 86, 27], qui offre des primitives simples et universelles pour exprimer des schémas de coordination et communication via une mémoire structurée.

Deuxième partie

Le Langage Circus

Objectifs Ce chapitre présente les choix fondamentaux à la base du méta-langage Circus et l'étude théorique des aspects les plus originaux.

1. **Orientations générales.** Circus se veut un langage spécialisé dans la transformation de structures. Afin de développer les possibilités d'intégration, Circus doit générer des *composants*, c'est à dire des pièces de logiciel dont l'interface est suffisamment explicite pour pouvoir être utilisées ou réutilisées dans des architectures variées. Le principe central est de reposer sur des structures de données suffisamment riche pour couvrir les besoins de modélisation impliqués dans le traitement des langages, qu'ils soient textuels ou visuels. Associées à ces structures, des transformations doivent permettre de spécifier les traitements nécessaires aux différents niveaux d'abstractions (lexical,syntaxique,sémantique).. Ces transformations peuvent être spécialisées lorsque les technologies sont éprouvées et suffisantes (analyse lexicale et syntaxique) ou plus générales afin de couvrir la variété des besoins. Dans tous les cas, les spécifications doivent prendre en compte la modularité et la réutilisation. Cette dernière passe par des mécanismes d'héritage mais aussi des facilités de composition. Les évolutions récentes permettent d'envisager l'analyse syntaxique, textuelle et visuelle dans le même cadre, les analyseurs utilisant des grammaires *pLR*. Sous certaines conditions, les spécifications de grammaires hors contexte peuvent être modulaires et réutilisables par héritage. La définition d'un formalisme adapté aux transformations générales dans les conditions envisagées est le point le plus difficile de cette étude. Le niveau d'abstraction doit être suffisamment élevé pour présenter un intérêt dans l'écriture d'outils de traitement de langages, mais de niveau suffisamment bas pour disposer d'une grande puissance expressive et d'une vision précise du contrôle d'exécution. Les points forts et les points faibles des formalismes existants ont été examinés dans le chapitre précédent. La réécriture de graphes manipule des structures de la richesse souhaitée mais pose des problèmes dans le contrôle de l'exécution. Les systèmes de réécriture de termes offrent la possibilité de définir des machines abstraites, lesquelles présentent de manière simple et directe la sémantique de l'exécution. Par contre, les structures manipulées manquent d'expressivité. L'approche de type *Centaure* semble de trop haut niveau pour répondre aux problèmes du contrôle de l'exécution, mais aussi, de l'accessibilité du langage au plus grand nombre possible d'utilisateurs. Nous explorons des machines abstraites d'un nouveau type, dites polymorphes, dans lesquelles il est possible de transformer une large classe de structures de données et qui apportent une réponse pour chacun des critères envisagés.
2. **Abstractions de données.** Les objectifs sont de favoriser l'expressivité et la réutilisation des schémas de données. De plus, les concepts proposés doivent rester les plus simples possibles, bien adaptés aux problèmes spécifiques de la transformation. Ainsi, la gestion de la mémoire doit être rendue transparente, et les abstractions doivent comporter des constructions universelles telles que listes, tables de hachage, ensembles, agrégats, et tuples. L'ensemble de ces abstractions est organisé autour d'un système de types offrant un polymorphisme par sous-typage.
3. **Abstractions de contrôle.** Nous avons vu que la maîtrise des mécanismes d'exécution était indispensable pour assurer l'universalité d'utilisation des composants de traitement. Soit pour les coupler à des schémas d' exécution et de communications externes, soit pour comprendre, mettre au point, ajuster les performances. Le concept de machines abstraites polymorphes propose un schéma d'exécution clair, adaptable aux structures de données devant être transformées. Par définition, elles spécifient des comportements complexes par succession d'étapes atomiques simples, faciles à comprendre, qui peuvent prendre la forme de règles. Ces dernières se composent d'une opération de filtrage (applicables sur l'ensemble des types de données du langage) en partie gauche et d'actions impératives (ou d'autres règles) en partie droite. Munies de ces mécanismes, nos machines

abstraites, semblent offrir le niveau d'abstraction idéal, suffisamment bas d'une part pour être général et explicites et d'autre part suffisamment élevé pour simplifier la tâche de spécification. Par ailleurs, elles offrent des possibilités compositionnelles qui favorisent la réutilisation mais aussi autorisent la conception d'architectures de traitement explicites et reconfigurables.

4. **Architectures.** La conception des architectures joue un rôle central dans la définition d'outils fortement intéractifs. En effet, ces derniers doivent s'adapter aux contraintes temporelles liées aux actions et réactions de l'utilisateur. De ce fait, les architectures "intéractives" sont traditionnellement bâties sur un modèle réactif, évènementiel. Les architectures de traitement classiques sont, elles, basées sur le modèle "batch processing" qui accepte des informations en entrée, les traite, puis délivre des résultats. D'une manière générale, un outil intéractif de type langage visuel est constitué d'une partie frontale, réactive, dans laquelle on tente de faire remonter le plus possible de sémantique, et d'une partie "arrière" (*back-end*), dans laquelle sont placées les transformations trop complexes pour pouvoir prendre part à un processus incrémental. Toutefois ces transformations peuvent également réaliser des vérifications et donc émettre des messages d'erreurs. Ces derniers doivent être transmis à l'utilisateur, si possible en tenant compte du contexte. Ainsi, même à ce niveau, le jeu des transformations entre les différents systèmes de représentation s'exerce encore. Seules les durées diffèrent : dans les couches frontales, les temps de réaction doivent rester en dessous de la seconde, dans les autres, les tolérances sont plus larges. De quelle façon un contrôleur de types doit il être construit pour pouvoir collaborer avec un éditeur syntaxique intéractif, ou s'intégrer dans un schéma de traitement plus classique ? De quelle façon conserver la modularité d'un contrôleur de types tout en le faisant collaborer avec le générateur de codes afin d'améliorer la qualité de l'optimisation ? Circus propose d'intégrer un modèle original de coordination, inspiré du langage *Linda* [27, 50], qui permet aux composants de traitement de communiquer et se coordonner au travers d'une mémoire structurée. Ainsi, il devient possible de concevoir les transformations de structures "interactives", en ce sens qu'elle tiennent compte de leur environnement, au travers d'un schéma de coordination, tout conservant l'indirection nécessaire à la réutilisation.

Le noyau du langage

Le noyau du langage *Circus* est essentiellement une extention du lambda-calcul typé non récursif, à laquelle nous avons adjoint des primitives permettant de construire des définitions modulaires. Un type énuméré original est proposé à ce niveau, qui permet d'utiliser au mieux les informations fournies par un programmeur lorsqu'il sait qu'une variable possède un nombre limité de valeurs licites. C'est précisément l'expérimentation qui nous a montré que ce cas se présentait très souvent dans le traitement des langages (et la transformation de structures en général), où l'on utilise abondament des attributs dont le domaine de valuation est fini pour décorer les arbres, graphes et autre structures.

Le langage est présenté formellement par sa syntaxe, sa sémantique opérationnelle et son système de types. Les sections suivantes permettront de caractériser le système de types (correction opérationnelle), puis de proposer des algorithmes pour l'interprétation et pour le contrôle des types.

4.1 Syntaxe

4.1.1 noyau fonctionnel

e	::=	**n** | **s** | **none** |	*litteraux*
		$e + e$ |	*addition / concatenation*
		if e_1 **then** e_2 **else** e_3 |	*choix*
		(e)	*parenthèsage*
		x	*variables*
e	::=	**true** | **false** |	*litteraux booléens*
		$e == e$ | $e \leq e$ |	*comparaisons*
		e **and** e | **not** e |	*connecteurs logiques*
e	::=	$\lambda x : t.e$ | $e(e)$	*lambda abstraction et application*

La syntaxe des types du langage est décrite par la grammaire suivante :

t	::=	τ |	*variable de type*
		$\bot$ | $\top$ |	*type minimal et maximal*
		num | **bool** | **string** | **None** |	*types primitifs*
		t **in** $\{e_1, \cdots, e_n\}$ |	*types enumérés*
		$t \rightarrow t$	*fonctions*

4.1.2 définitions modulaires

c	::=	**module** x { d } | $c\ c$	*espace(s) de définition*
d	::=	**const** $x : t = e$ |	*constante typée*
		const $x := e$ |	*constante avec type inféré*
		type $\tau = t$ |	*définition de types nommés*
		from x **import** x **as** x |	*importation de définitions externes*
		$d\ d$	

4.2 Sémantique opérationnelle

4.2.1 Introduction

La réduction des termes, ou "calcul" est définie au moyen d'une relation de transition $\rightarrow$ qui exprime l'évolution temporelle des termes, associée à un contexte $\mathcal{S}$, espace mémoire contenant les définitions constantes et variables de l'interpréteur. Les termes syntaxiques décrits par la grammaire de *Circus*, notés c_i, d_i, e_i, en relation avec les non-terminaux de la grammaire hors-contexte, sont réduits après évaluations dans des termes "normaux" (ne pouvant plus être réduits en autre chose qu'eux-mêmes) notés v_i, w_i. La réduction des types, notés t_i, en types "normaux" notés u_i, est également introduite au moyen de la même relation. La relation de transition est fermée en ce sens qu'elle transforme tout terme du langage décrit par la grammaire en un autre terme du même langage. Toutefois, l'ensemble des termes engendrés par la grammaire de *Circus* est augmenté d'un élément, **error**, qui dénote une erreur d'exécution. Aucun type n'est associé à cet élément du langage. Bien que cette valeur distinguée ne puisse se transformer en aucune autre, elle n'est pas considérée comme normale, en ce sens qu'aucune règle ne permet de la réécrire en elle-même. La relation $\twoheadrightarrow$ est la fermeture transitive de $\rightarrow$:

$$\frac{[\mathcal{S} \vdash e] \rightarrow [\mathcal{S}' \vdash e']}{[\mathcal{S} \vdash e] \twoheadrightarrow [\mathcal{S}' \vdash e']}\ [\text{e-dir}]$$

$$\frac{[\mathcal{S} \vdash e] \twoheadrightarrow [\mathcal{S}' \vdash e'] \quad [\mathcal{S}' \vdash e'] \twoheadrightarrow [\mathcal{S}'' \vdash e'']}{[\mathcal{S} \vdash e'] \twoheadrightarrow [\mathcal{S}'' \vdash e'']}\ [\text{e-trans}]$$

Une autre relation, notée $\rightarrow\circ$ représente la réduction aboutissant à une forme normale, ne pouvant plus être transformée en un autre terme que lui-même.

$$\frac{[\mathcal{S} \vdash e] \rightarrow [\mathcal{S}' \vdash v] \quad [\mathcal{S}' \vdash v] \rightarrow [\mathcal{S}' \vdash v]}{[\mathcal{S} \vdash e] \rightarrow\circ\ [\mathcal{S}' \vdash v]}\ [\text{e-norm}]$$

la relation $\twoheadrightarrow\circ$ est la variante correspondant à l'aboutissement d'une succession d'étapes

$$\frac{[\mathcal{S} \vdash e] \twoheadrightarrow [\mathcal{S}' \vdash v] \quad [\mathcal{S}' \vdash v] \rightarrow [\mathcal{S}' \vdash v]}{[\mathcal{S} \vdash e] \twoheadrightarrow\circ\ [\mathcal{S}' \vdash v]}\ [\text{e-norm-t}]$$

4.2.2 Environnement d'exécution

L'environnement est une structure d'exécution permettant de stocker la valeur des constantes et variables du langage ($\mathcal{S}$).

L'ensemble des environnements que nous considérons dans la suite de ce développement est noté $\mathbb{S}$. Ceux-ci mémorisent des couples (nom logique, valeur), notés $x = v$ où les noms logiques x_i sont uniques dans $\mathcal{S}$, et les valeurs associées v_i sont des termes ou des types normalisés du langage. Un environnement $\mathcal{S}$ peut aussi contenir des couples $x = \mathcal{S}'$, ou $\mathcal{S}'$ est un environnement distinct de $\mathcal{S}$.

$\mathcal{S}(x_i)$ retourne v_i si $x_i = v_i \in \mathcal{S}$. La notation $\mathcal{S}, x = v$ est une facilité d'écriture pour $\mathcal{S} \cup \{x = v\}$. De même, $\mathcal{S}, \mathcal{S}'$ est un raccourci d'écriture pour $\mathcal{S} \cup \mathcal{S}'$, ou $\mathcal{S}$ et $\mathcal{S}'$ sont disjoints.

4.2.3 Le système de transition : calcul des termes

Les premières règles expriment la réduction des variables de termes et de type dans l'environnement d'exécution. L'évaluation de variables non mémorisées dans l'environnement provoque une erreur.

$$[\mathcal{S}, x = v \vdash x] \rightarrow [\mathcal{S}, x = v \vdash v] \quad \text{[e-var]}$$

$$x \notin \text{dom}(\mathcal{S}) \quad [\mathcal{S} \vdash x] \rightarrow [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-var-err]}$$

$$[\mathcal{S}, \tau = t \vdash \tau] \rightarrow [\mathcal{S}, \tau = t \vdash t] \quad \text{[e-tvar]}$$

$$\tau \notin \text{dom}(\mathcal{S}) \quad [\mathcal{S} \vdash \tau] \rightarrow [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-tvar-err]}$$

Les termes "primitifs" sont déjà sous forme normale, ce qui est formalisé par :

$$[\mathcal{S} \vdash v] \rightarrow [\mathcal{S} \vdash v] \quad \text{[e-base]} \quad v \in \{\textbf{n}, \textbf{s}, \textbf{false}, \textbf{true}, \textbf{none}, \textbf{unit}\}$$

Le traitement de l'alternative est sans surprise. Notons toutefois que l'évaluation de la partie booléene est susceptible de modifier le contexte $\mathcal{S}$ (cf [e-if]).

$$\frac{[\mathcal{S} \vdash e_1] \rightarrow [\mathcal{S}' \vdash e_1']}{[\mathcal{S} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S}' \vdash \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3]} \quad \text{[e-if]}$$

$$[\mathcal{S} \vdash \textbf{if true then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S} \vdash e_2] \quad \text{[e-if1]}$$

$$[\mathcal{S} \vdash \textbf{if false then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S} \vdash e_3] \quad \text{[e-if2]}$$

$$[\mathcal{S} \vdash \textbf{if error then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-if-err]}$$

Les règles suivantes fixent l'ordre d'évaluation lors d'un appel de fonction. D'autre part, l'appel proprement dit est "classiquement" calculé au moyen d'une béta réduction, où la valeur d'appel est substituée au paramètre formel dans le corps de la lambda fonction (cf [e-β]). La valeur distinguée **error** est propagée de manière homogène, comme dans l'ensemble des évaluations. Notons également qu'une lambda-expression est normalisée quand le type de son argument d'entrée est normalisé (cf [e-lam]).

$$\frac{[\mathcal{S} \vdash t] \twoheadrightarrow\circ\ \ [\mathcal{S} \vdash u]}{[\mathcal{S} \vdash \lambda x : t.e] \to [\mathcal{S} \vdash \lambda x : u.e]}\ \text{[e-lam]}$$

$$\frac{[\mathcal{S} \vdash e_2] \to [\mathcal{S} \vdash e_2']}{[\mathcal{S} \vdash e_1(e_2)] \to [\mathcal{S} \vdash e_1(e_2')]}\ \text{[e-@1]}$$

$$\frac{[\mathcal{S} \vdash e_1] \to [\mathcal{S} \vdash e_1']}{[\mathcal{S} \vdash e_1(v_2)] \to [\mathcal{S} \vdash e_1'(v_2)]}\ \text{[e-@2]}$$

$$[\mathcal{S} \vdash \lambda x : t.e_3(v_2)] \to [\mathcal{S} \vdash e_3[v_2/x]]\ \ \text{[e-}\beta\text{]}$$

$$[\mathcal{S} \vdash e_1(\textbf{error})] \to [\mathcal{S} \vdash \textbf{error}]\ \ \text{[e-@-err1]}$$

$$[\mathcal{S} \vdash \textbf{error}(v_2)] \to [\mathcal{S} \vdash \textbf{error}]\ \ \text{[e-@-err2]}$$

$$\frac{v_1 \not\equiv \lambda x : t.e}{[\mathcal{S} \vdash v_1(v_2)] \to [\mathcal{S} \vdash \textbf{error}]}\ \text{[e-@-err3]}$$

Un module contient une série de définitions, constantes ou types. Ce premier est évalué à **unit** lorsque l'ensemble de ces définitions est lui-même évalué à **unit**. Dans ce cas, le contexte original $\mathcal{S}$ a été augmenté de $\mathcal{S}'$, contenant le résultat de toutes les évaluations réalisées dans le corps du module. Notons l'utilisation de la relation $[\ \vdash\ ] \twoheadrightarrow\circ\ \ [\ \vdash\ ]$ dans le numérateur de [e-mod], qui exprime que ces actions se sont réalisées en de multiples transitions élémentaires. Ce contexte $\mathcal{S}'$ est alors inséré dans le contexte d'origine sous le nom du module, x. Les règles [e-mod-all] décrivent simplement l'enchaînement du traitement de plusieurs modules successifs.

$$\frac{[\mathcal{S} \vdash d] \twoheadrightarrow\circ\ \ [\mathcal{S}, \mathcal{S}' \vdash \textbf{unit}]}{[\mathcal{S} \vdash \textbf{module}\ x\{d\}] \to [\mathcal{S}, x = \mathcal{S}' \vdash \textbf{unit}]}\ \text{[e-mod]}$$

$$\frac{[\mathcal{S} \vdash d] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{error}]}{[\mathcal{S} \vdash \textbf{module}\ x\{d\}] \to [\mathcal{S}' \vdash \textbf{error}]}\ \text{[e-mod-err]}$$

$$\frac{[\mathcal{S} \vdash c_1] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{unit}] \quad [\mathcal{S}' \vdash c_2] \twoheadrightarrow [\mathcal{S}'' \vdash \textbf{unit}]}{[\mathcal{S} \vdash c_1\ c_2] \to [\mathcal{S}'' \vdash \textbf{unit}]}\ \text{[e-all-mod]}$$

$$\frac{[\mathcal{S} \vdash c_1] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{error}]}{[\mathcal{S} \vdash c_1\ c_2] \to [\mathcal{S}' \vdash \textbf{error}]}\ \text{[e-all-mod-err2]}$$

$$\frac{[\mathcal{S} \vdash c_1] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{unit}] \quad [\mathcal{S}' \vdash c_2] \twoheadrightarrow [\mathcal{S}'' \vdash \textbf{error}]}{[\mathcal{S} \vdash c_1\ c_2] \to [\mathcal{S}' \vdash \textbf{error}]}\ \text{[e-all-mod-err2]}$$

Les définitions de constantes et types internes aux modules sont calculées puis insérées dans le contexte d'exécution, quand aucune erreur n'apparait au cours de l'évaluation.

$$\frac{[\mathcal{S} \vdash e] \twoheadrightarrow\!\circ\ \ [\mathcal{S} \vdash v]}{[\mathcal{S} \vdash \textbf{const}\ x : t = e] \to [\mathcal{S}, x = v \vdash \textbf{unit}]}\ \text{[e-def]}$$

$$\frac{[\mathcal{S} \vdash e] \twoheadrightarrow\!\circ\ \ [\mathcal{S}' \vdash \textbf{error}]}{[\mathcal{S} \vdash \textbf{const}\ x : t = e] \to [\mathcal{S}' \vdash \textbf{error}]}\ \text{[e-def-err]}$$

$$\frac{[\mathcal{S} \vdash e] \twoheadrightarrow\!\circ\ \ [\mathcal{S} \vdash v]}{[\mathcal{S} \vdash \textbf{const}\ x := e] \to [\mathcal{S}, x = v \vdash \textbf{unit}]}\ \text{[e-idef]}$$

$$\frac{[\mathcal{S} \vdash e] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{error}]}{[\mathcal{S} \vdash \textbf{const}\ x := e] \to [\mathcal{S}' \vdash \textbf{error}]}\ \text{[e-idef-err]}$$

$$\frac{[\mathcal{S} \vdash t] \twoheadrightarrow\!\circ\ \ [\mathcal{S} \vdash u]}{[\mathcal{S} \vdash \textbf{type}\ \tau = t] \to [\mathcal{S}, \tau = u \vdash \textbf{unit}]}\ \text{[e-tdef]}$$

$$\frac{[\mathcal{S} \vdash t] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{error}]}{[\mathcal{S} \vdash \textbf{type}\ \tau = t] \to [\mathcal{S}' \vdash \textbf{error}]}\ \text{[e-tdef-err]}$$

Les importations de définitions externes (définies dans d'autres modules que le module courant) utilisent la structure "emboîtée" des contextes (cf définition d'un module [e-mod]). Naturellement le nom du module doit être présent dans le contexte d'exécution (cf [e-imp-err]), mais aussi la définition concernée ([e-imp-err2]). Notons toutefois que le nouveau nom donné à la référence importée peut "écraser" une référence déjà présente dans le contexte d'exécution (aucune condition n'est attachée à x_3 dans [e-imp]). Les autres cas d'erreur sont les tentatives d'importation de références appliquées à d'autres structures que des modules ([e-imp-err3] et [e-imp-err4])

$$\frac{x_2 \in \text{dom}(\mathcal{S}')}{[\mathcal{S}, x_1 = \mathcal{S}' \vdash \textbf{from}\ x_1\ \textbf{import}\ x_2\ \textbf{as}\ x_3] \to [\mathcal{S}, x_1 = \mathcal{S}', x_3 = \mathcal{S}'(x_2) \vdash \textbf{unit}]}\ \text{[e-imp]}$$

$$\frac{x_1 \notin \text{dom}(\mathcal{S})}{[\mathcal{S} \vdash \textbf{from}\ x_1\ \textbf{import}\ x_2\ \textbf{as}\ x_3] \to [\mathcal{S} \vdash \textbf{error}]}\ \text{[e-imp-err]}$$

$$\frac{x_2 \notin \text{dom}(\mathcal{S}')}{[\mathcal{S}, x_1 = \mathcal{S}' \vdash \textbf{from}\ x_1\ \textbf{import}\ x_2\ \textbf{as}\ x_3] \to [\mathcal{S}, x_1 = \mathcal{S}' \vdash \textbf{error}]}\ \text{[e-imp-err2]}$$

$$[\mathcal{S}, x_1 = v \vdash \textbf{from}\ x_1\ \textbf{import}\ x_2\ \textbf{as}\ x_3] \to [\mathcal{S}, x_1 = v \vdash \textbf{error}]\ \ \text{[e-imp-err3]}$$

$$[\mathcal{S}, x_1 = u \vdash \textbf{from}\ x_1\ \textbf{import}\ x_2\ \textbf{as}\ x_3] \to [\mathcal{S}, x_1 = u \vdash \textbf{error}]\ \ \text{[e-imp-err4]}$$

Enfin, [e-all-def] décrit simplement la succession des traitements pour toutes les définitions d'un module, et [e-all-def-err] la propagation des erreurs (arrêt du traitement à la première erreur rencontrée)

$$\frac{[\mathcal{S} \vdash d_1] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{unit}] \quad [\mathcal{S}' \vdash d_2] \twoheadrightarrow [\mathcal{S}'' \vdash \textbf{unit}]}{[\mathcal{S} \vdash d_1\ d_2] \to [\mathcal{S}'' \vdash \textbf{unit}]}\ \text{[e-all-def]}$$

$$[\mathcal{S} \vdash \textbf{error}\ d_2] \to [\mathcal{S} \vdash \textbf{error}]\ \ \text{[e-all-def-err]}$$

Le comportement des connectives logiques est défini sans surprise, avec priorité à l'opérande gauche dans l'évaluation du **and**. Notons également le comportement "absorbant" de la valeur **none**.

$$\frac{[\mathcal{S} \vdash e_1] \to [\mathcal{S}' \vdash e_1']}{[\mathcal{S} \vdash e_1 \textbf{ and } e_2] \to [\mathcal{S}' \vdash e_1' \textbf{ and } e_2]} \text{ [e-and1]}$$

$$[\mathcal{S} \vdash \textbf{true and } e_2] \to [\mathcal{S} \vdash e_2] \text{ [e-and2]}$$

$$[\mathcal{S} \vdash \textbf{false and } e_2] \to [\mathcal{S} \vdash \textbf{false}] \text{ [e-and3]}$$

$$[\mathcal{S} \vdash \textbf{none and } e_2] \to [\mathcal{S} \vdash \textbf{none}] \text{ [e-and4]}$$

$$\frac{[\mathcal{S} \vdash e] \to\!\!\circ\ [\mathcal{S}' \vdash e']}{[\mathcal{S} \vdash \textbf{not } e] \to [\mathcal{S}' \vdash \textbf{not } e']} \text{ [e-not]}$$

$$[\mathcal{S} \vdash \textbf{not false}] \to [\mathcal{S} \vdash \textbf{true}] \text{ [e-not1]}$$

$$[\mathcal{S} \vdash \textbf{not true}] \to [\mathcal{S} \vdash \textbf{false}] \text{ [e-not2]}$$

$$[\mathcal{S} \vdash \textbf{not none}] \to [\mathcal{S} \vdash \textbf{none}] \text{ [e-not3]}$$

[e-op1] et [e-op2] sont des équations générales qui fixent une priorité à l'opérande gauche pour les opération binaires $+, ==, \leq$.

$$\frac{[\mathcal{S} \vdash e_1] \to [\mathcal{S}' \vdash e_1']}{[\mathcal{S} \vdash e_1 \star e_2] \to [\mathcal{S}' \vdash e_1' \star e_2]} \text{ [e-op1]} \quad (\star \in \{+, ==, \leq\})$$

$$\frac{[\mathcal{S} \vdash e_2] \to [\mathcal{S}' \vdash e_2']}{[\mathcal{S} \vdash v_1 \star e_2] \to [\mathcal{S}' \vdash v_1 \star e_2']} \text{ [e-op2]} \quad (\star \in \{+, ==, \leq\})$$

Dans les équations suivantes, la fonction $\epsilon()$ et son inverse $\epsilon^{-1}()$ permet de transformer une unité lexicale numérique en un numérique (comme par exemple $\epsilon(\text{"0123"}) = 123$), et réciproquement pour son inverse $\epsilon^{-1}()$. L'addition de valeurs numériques possède la sémantique additive classique, et l'addition de chaînes de caractères possède la sémantique de concaténation. Les booléens autres valeurs hétérogènes ne peuvent pas être additionnés entre eux, ni à gauche ni à droite (de [e-plus-err1] à [e-plus-err4]).

$$\frac{n_3 = \epsilon^{-1}(\epsilon(n_1) + \epsilon(n_2))}{[\mathcal{S} \vdash n_1 + n_2] \to [\mathcal{S} \vdash n_3]} \text{ [e-plus-n]}$$

$$[\mathcal{S} \vdash s_1 + s_2] \to [\mathcal{S} \vdash s_1 s_2] \text{ [e-plus-s]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash n + v] \to [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-plus-err1]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash v + n] \to [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-plus-err2]}$$

$$\frac{v \in \{\mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash s + v] \to [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-plus-err3]}$$

$$\frac{v \in \{\mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash v + s] \to [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-plus-err4]}$$

Les comparaisons de valeurs numériques possèdent également la sémantique attendue.

$$\frac{\epsilon(n_1) \leq \epsilon(n_2)}{[\mathcal{S} \vdash n_1 \leq n_2] \to [\mathcal{S} \vdash \mathbf{true}]} \text{ [e-ineq-num1]}$$

$$\frac{\epsilon(n_1) \not\leq \epsilon(n_2)}{[\mathcal{S} \vdash n_1 \leq n_2] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-ineq-num2]}$$

La comparaison de chaînes correspond à la présence d'une sous-chaîne en tête. Elle n'utilise donc pas l'ordre lexicographique habituel, qui permet par exemple d'évaluer à vrai l'expression 'abc' $\leq$ 'ad'.

$$\frac{s_2 \equiv s_1 s_3}{[\mathcal{S} \vdash s_1 \leq s_2] \to [\mathcal{S} \vdash \mathbf{true}]} \text{ [e-ineq-str1]}$$

$$\frac{s_2 \not\equiv s_1 s_3}{[\mathcal{S} \vdash s_1 \leq s_2] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-ineq-str2]}$$

L'ordre des valeur booléennes est tel que **false** est plus petit que **true**.

$$[\mathcal{S} \vdash \mathbf{false} \leq \mathbf{true}] \to [\mathcal{S} \vdash \mathbf{true}] \text{ [e-ineq-bool1]}$$

$$[\mathcal{S} \vdash \mathbf{false} \leq \mathbf{false}] \to [\mathcal{S} \vdash \mathbf{true}] \text{ [e-ineq-bool2]}$$

$$[\mathcal{S} \vdash \mathbf{true} \leq \mathbf{true}] \to [\mathcal{S} \vdash \mathbf{true}] \text{ [e-ineq-bool3]}$$

$$[\mathcal{S} \vdash \mathbf{true} \leq \mathbf{false}] \to [\mathcal{S} \vdash \mathbf{false}] \text{ [e-ineq-bool4]}$$

Enfin les cas d'erreur correspondent à la comparaison de valeurs ayant des types incompatibles.

$$\frac{v \in \{\mathbf{s}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash n \leq v] \rightarrow [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-ineq-err1]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash v \leq n] \rightarrow [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-ineq-err2]}$$

$$\frac{v \in \{\mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash s \leq v] \rightarrow [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-ineq-err3]}$$

$$\frac{v \in \{\mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash v \leq s] \rightarrow [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-ineq-err4]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{n}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash b \leq v] \rightarrow [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-ineq-err5]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{n}, \mathbf{unit}, \lambda x : t.e\}}{[\mathcal{S} \vdash v \leq b] \rightarrow [\mathcal{S} \vdash \mathbf{error}]} \text{ [e-ineq-err6]}$$

$$[\mathcal{S} \vdash \lambda x : t_1 . e_1 \leq \lambda y : t_2 . e_2] \rightarrow [\mathcal{S} \vdash \mathbf{error}] \text{ [e-ineq-err7]}$$

Le test d'égalité est plus souple que l'inégalité, car il ne produit jamais d'érreurs, même si les valeurs sont incompatibles. Notons que l'égalité de deux lambda fonctions tient compte des différences potentielles dans les noms des paramètres formels (on applique l'*alpha conversion*), mais réclame l'identité syntaxique sur les types des paramètres.

$$\frac{v_1, v_2 \in \{\mathbf{s}, \mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{none}, \mathbf{unit}\} \quad v_1 \equiv v_2}{[\mathcal{S} \vdash v_1 == v_2] \to [\mathcal{S} \vdash \mathbf{true}]} \text{ [e-eq1]}$$

$$\frac{v_1, v_2 \in \{\mathbf{s}, \mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{none}, \mathbf{unit}\} \quad v_1 \not\equiv v_2}{[\mathcal{S} \vdash v_1 == v_2] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-eq2]}$$

$$\frac{e_1 \equiv e_2[x/y]}{[\mathcal{S} \vdash \lambda x : u.e_1 == \lambda y : u.e_2] \to [\mathcal{S} \vdash \mathbf{true}]} \text{ [e-eq-lamb1]}$$

$$\frac{u_1 \not\equiv u_2}{[\mathcal{S} \vdash \lambda x : u_1.e_1 == \lambda y : u_2.e_2] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-eq-lamb2]}$$

$$\frac{e_1 \not\equiv e_2[x/y]}{[\mathcal{S} \vdash \lambda x : u.e_1 == \lambda y : u.e_2] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-eq-lamb3]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{none}, \mathbf{unit}\}}{[\mathcal{S} \vdash \lambda x : t_1.e_1 == v] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-eq-lamb4]}$$

$$\frac{v \in \{\mathbf{s}, \mathbf{n}, \mathbf{true}, \mathbf{false}, \mathbf{none}, \mathbf{unit}\}}{[\mathcal{S} \vdash v == \lambda x : t_1.e_1] \to [\mathcal{S} \vdash \mathbf{false}]} \text{ [e-eq-lamb5]}$$

Les valeurs **none** et **error** se comportent de manière similaire, en ce sens qu'elles sont toutes deux "absorbantes".

$$[\mathcal{S} \vdash \mathbf{error} \star e] \to [\mathcal{S} \vdash \mathbf{error}] \text{ [e-err-op1]}$$

$$[\mathcal{S} \vdash e \star \mathbf{error}] \to [\mathcal{S} \vdash \mathbf{error}] \text{ [e-err-op2]}$$

$$\star \in \{+, ==, \leq\} \quad [\mathcal{S} \vdash \mathbf{none} + e] \to [\mathcal{S} \vdash \mathbf{none}] \text{ [e-none-op1]}$$

$$[\mathcal{S} \vdash v + \mathbf{none}] \to [\mathcal{S} \vdash \mathbf{none}] \text{ [e-none-op2]}$$

$$[\mathcal{S} \vdash \mathbf{not\ error}] \to [\mathcal{S} \vdash \mathbf{error}] \text{ [e-err-not]}$$

Le traitement des valeurs **none** dans les opérations consiste donc à propager la valeur **none**, comme un élement absorbant. Une autre approche aurait été de déclarer une erreur d'exécution. Cependant, la valeur **none** est un terme possédant tous les types licites, il serait donc incohérent de générer une erreur d'exécution pour une expression bien typée, telle que

$$\frac{\emptyset \rhd 10 : \mathbf{num} \quad \dfrac{\emptyset \rhd \mathbf{none} : \bot \quad \dfrac{\emptyset \rhd \mathbf{num}}{\emptyset \rhd \bot \preccurlyeq \mathbf{num}} \text{ [st-bot]}}{\emptyset \rhd \mathbf{none} : \mathbf{num}} \text{ [st-sub]}}{\emptyset \rhd 10 + \mathbf{none} : \mathbf{num}} \text{ [te-plus-num]}$$

Enfin, il nous reste à définir l'égalité (et l'inégalité) lorsque une au moins des valeurs est **none**.

$$[\mathcal{S} \vdash \textbf{none} == \textbf{none}] \to [\mathcal{S} \vdash \textbf{true}] \quad \text{[e-eq-none]}$$

$$[\mathcal{S} \vdash \textbf{none} \leq \textbf{none}] \to [\mathcal{S} \vdash \textbf{true}] \quad \text{[e-ineq-none]}$$

$$[\mathcal{S} \vdash \textbf{none} \leq v] \to [\mathcal{S} \vdash \textbf{true}] \quad \text{[e-ineq-none2]}$$

$$v \not\equiv \textbf{none} \quad [\mathcal{S} \vdash v \leq \textbf{none}] \to [\mathcal{S} \vdash \textbf{false}] \quad \text{[e-ineq-none3]}$$

$$[\mathcal{S} \vdash \textbf{if none then } e_2 \textbf{ else } e_3] \to [\mathcal{S} \vdash \textbf{none}] \quad \text{[e-if-none]}$$

4.2.4 Le système de transition : calcul des types

Les expressions de types se réduisent de la manière suivante :

1. Les types de base sont déjà normaux.

$$[\mathcal{S} \vdash u] \to [\mathcal{S} \vdash u] \quad \text{[e-tbase]} \quad u \in \{\textbf{num}, \textbf{string}, \top, \bot, \textbf{None}, \textbf{Unit}, \textbf{bool}\}$$

2. les types *fonction* sont normalisés quand les deux sous-types le sont ([e-lam]). Dans le cas contraire, on normalise d'abord le sous-type de gauche puis le sous-type de droite ([e-tlam1] et [e-tlam2])

$$\frac{[\mathcal{S} \vdash t_1] \to [\mathcal{S} \vdash t_1']}{[\mathcal{S} \vdash t_1 \to t_2] \to [\mathcal{S} \vdash t_1' \to t_2]} \text{ [e-tlam1]}$$

$$\frac{[\mathcal{S} \vdash t_2] \to [\mathcal{S} \vdash t_2']}{[\mathcal{S} \vdash u_1 \to t_2] \to [\mathcal{S} \vdash u_1 \to t_2']} \text{ [e-tlam2]}$$

$$[\mathcal{S} \vdash u_1 \to u_2] \to [\mathcal{S} \vdash u_1 \to u_2] \quad \text{[e-tlam]}$$

$$[\mathcal{S} \vdash \textbf{error} \to t_2] \to [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-tlam-err1]}$$

$$[\mathcal{S} \vdash u_1 \to \textbf{error}] \to [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-tlam-err2]}$$

3. les types *énumeration* sont normalisés lorsque le type "principal" est normal, et que tous les termes constituant l'énumération le sont ([e-tenum]). Sinon, le type principal est d'abord réduit, puis tous les termes, dans l'ordre d'occurence.

$$\frac{[\mathcal{S} \vdash t] \to [\mathcal{S} \vdash t']}{[\mathcal{S} \vdash t \textbf{ in } \{e_1, \cdots, e_n\}] \to [\mathcal{S} \vdash t' \textbf{ in } \{e_1, \cdots, e_n\}]} \text{ [e-tenum]}$$

$$\frac{[\mathcal{S} \vdash e_i] \to [\mathcal{S} \vdash e_i']}{[\mathcal{S} \vdash u \textbf{ in } \{u_1, \cdots, e_i, \cdots, e_n\}] \to [\mathcal{S} \vdash u \textbf{ in } \{u_1, \cdots, e_i', \cdots, e_n\}]} \text{ [e-tenum2]}$$

$$[\mathcal{S} \vdash u \textbf{ in } \{u_1, \cdots, u_n\}] \to [\mathcal{S} \vdash u \textbf{ in } \{u_1, \cdots, u_n\}] \quad \text{[e-tenum]}$$

$$[\mathcal{S} \vdash \textbf{error in } \{e_1, \cdots, e_n\}] \to [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-tenum-err1]}$$

$$[\mathcal{S} \vdash u \textbf{ in } \{v_1, \cdots, \textbf{error}, \cdots, e_n\}] \to [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-tenum-err2]}$$

4.3 Système de types

4.3.1 Environnement de typage et jugements

L'environnement de typage, noté γ, ou γ' est un ensemble contenant des couples, notés $x : t = v$, $x : t$ ou $\tau :: t$. Le premier couple a pour signification informelle "la variable x représente un terme bien typé de type t, ayant pour valeur v". Le second signifie "la variable x représente un terme bien typé de type t", et le dernier "la variable τ représente un type bien formé de valeur t". Les jugements sont de la forme $\gamma \ \rhd \ t$ (le type t est bien formé dans l'environnement γ), $\gamma \rhd e : t$ (le terme e a pour type t dans l'environnement γ) et $\gamma \rhd t_1 \preccurlyeq t_2$ (le type t_1 est un sous-type de t_2 dans γ). Les équations suivantes permettent d'extraire les informations pertinentes de l'environnement.

$$\gamma, x : t = \rhd x : t \quad \text{[te-env]}$$

$$\gamma, x : t = v \rhd x : t \quad \text{[te-env2]}$$

$$\frac{\gamma, \tau :: t \rhd e : t}{\gamma, \tau :: t \rhd e : \tau} \quad \text{[te-env3]}$$

$$\frac{\gamma, \tau :: t \rhd t}{\gamma, \tau :: t \rhd \tau} \quad \text{[t-env]}$$

$$\frac{\gamma, \tau :: t \rhd t \preccurlyeq t_2}{\gamma, \tau :: t \rhd \tau \preccurlyeq t_2} \quad \text{[st-env1]}$$

$$\frac{\gamma, \tau :: t \rhd t_1 \preccurlyeq t}{\gamma, \tau :: t \rhd t_1 \preccurlyeq \tau} \quad \text{[st-env2]}$$

De plus, un environnement peut lui-même contenir des couples $x_1 :: \gamma'$, où γ' est un autre environnement. Le domaine d'un environnement γ, noté $\text{dom}(\gamma)$ est l'ensemble de tous les noms logiques qu'il contient.

Définition 4.1 *domaine d'un environnement*

(1) $x \in$ *dom*(γ) *ssi* $x : t = v \in \gamma$ *ou* $x : t \in \gamma$ *ou* $x :: \gamma' \in \gamma$

(2) $\tau \in$ *dom*(γ) *ssi* $\tau :: t \in \gamma$

4.3.2 Types bien formés

$$\emptyset \rhd \textbf{num} \quad \emptyset \rhd \textbf{bool} \quad \emptyset \rhd \textbf{string}$$

$$\emptyset \rhd \bot \qquad \emptyset \rhd \top \qquad \emptyset \rhd \textbf{None} \quad \emptyset \rhd \textbf{Unit}$$

$$\frac{\gamma \ \rhd \ t_1 \quad \gamma \ \rhd \ t_2}{\gamma \ \rhd \ t_1 \rightarrow t_2} \quad \text{[t-lamb]}$$

$$1 \leq i \leq n, \exists v_i \text{ tels que } \quad \frac{\gamma \ \rhd \ t \quad \gamma \rhd e_i : t \quad \gamma \rhd e_i \rightsquigarrow v_i}{\gamma \ \rhd \ t \ \textbf{in} \ \{e_1, \cdots, e_n\}} \quad \text{[t-enum]}$$

L'équation [t-enum] utilise un jugement $\gamma \rhd e \rightsquigarrow v$ qui assure que le terme e est calculable et converge vers une valeur unique v. Le noyau de langage proposé dans cette première partie possède la propriété

d'avoir une sémantique opérationnelle confluente, en ce sens que les expressions sont évaluables en un nombre fini d'étapes, et de manière déterministe. Avec les extensions proposées par la suite, un test syntaxique *conv*(γ, e) permettra de garantir cette propriété pour tout terme e bien typé (elle n'est pas décidable de manière générale). On peut écrire

$$\exists v \quad \frac{\gamma \rhd e : \top \quad \mathit{conv}(\gamma, e)}{\gamma \rhd e \rightsquigarrow v} \text{ [t-conv]}$$

afin de formaliser cette propriété ($\gamma \rhd e \rightsquigarrow v$ signifie donc "le terme e est évaluable en un terme normal unique v,dans l'environnement γ"). Nous définissons cette notion formellement par

Définition 4.2 *terme confluent*

$$\gamma \rhd e \rightsquigarrow v \quad \Leftrightarrow \quad \begin{cases} \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash e] \twoheadrightarrow\!\circ \ [\mathcal{S}' \vdash v] \\ ([\mathcal{S} \vdash e] \rightarrow [\mathcal{S}'' \vdash e'] \Rightarrow \gamma \rhd e' \rightsquigarrow v) \end{cases} \quad [\textit{te-conv}]$$

La notation $\Vdash \mathcal{S} : \gamma$ désigne tout environnement d'exécution $\mathcal{S}$ *conforme* au contexte de typage γ, c'est à dire contenant les mêmes constantes, et dont le type est de plus conforme. Cette notion sera formellement présentée dans la suite du développement.

4.3.3 Termes correctement typés

Nous produisons d'abord les axiomes, qui établissent le typage des unités lexicales du langage.

$$\begin{array}{cc} \emptyset \rhd \mathbf{n} : \mathbf{num} & \emptyset \rhd \mathbf{true} : \mathbf{bool} \\ \emptyset \rhd \mathbf{false} : \mathbf{bool} & \emptyset \rhd \mathbf{s} : \mathbf{string} \\ \emptyset \rhd \mathbf{none} : \mathbf{None} & \emptyset \rhd \mathbf{unit} : \mathbf{Unit} \end{array}$$

La première remarque est relative au type minimal $\bot$: aucun terme du langage ne possède ce type qui est "technique". Dans certaines formalisations de systèmes de types, $\bot$ dénote de ce fait une erreur de typage. Ce n'est pas systématiquement le cas dans Circus, comme nous le verrons par la suite. La deuxième remarque concerne $\top$: aucun terme n'est directement de ce type. Les termes bien typés possèdent le type maximal par le biais d'une relation de sous-typage qui sera présentée plus loin.

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd t_1 \rightarrow t_2 \quad \gamma, x : t_1 \rhd e : t_2}{\gamma \rhd \lambda x : t_1 . e : t_1 \rightarrow t_2} \text{ [te-lamb]}$$

$$\frac{\gamma \rhd e_1 : t_1 \rightarrow t_2 \quad \gamma \rhd e_2 : t_1}{\gamma \rhd e_1(e_2) : t_2} \text{ [te-@]}$$

$$\gamma \rhd \mathbf{None} \preccurlyeq t \quad \frac{\gamma \rhd e_1 : \mathbf{bool} \quad \gamma \rhd e_2 : t \quad \gamma \rhd e_3 : t}{\gamma \rhd \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : t} \text{ [te-if]}$$

Ces définitions sont "classiques" en se sens qu'on les trouve telles quelles dans de nombreux langages fonctionnels. Le typage des tests n'est pas complet, en ce sens que les types booléens construits sur des énumérations donnent des assertions de typage plus fine. Pour capturer cette expressivité, nous proposons les équations suivantes

$$\gamma \rhd \textbf{None} \preccurlyeq t \quad \frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 : t \quad \gamma \rhd e_3 : t}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t} \text{ [te-if1]}$$

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e_2 : t \quad \gamma \rhd e_3 : t}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t} \text{ [te-if2]}$$

$$\gamma \rhd \textbf{None} \preccurlyeq t_1 \quad \frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{true}, \textbf{none}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t_1} \text{ [te-if3]}$$

$$\gamma \rhd \textbf{None} \preccurlyeq t_2 \quad \frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t_2} \text{ [te-if4]}$$

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{true}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t_1} \text{ [te-if-true]}$$

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{false}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t_2} \text{ [te-if-false]}$$

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{none}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \textbf{None}} \text{ [te-if-none]}$$

Les définitions suivantes introduisent précisément le type énuméré, associé à une relation d'équivalence sur les termes

$$\frac{\gamma \rhd t \textbf{ in } \{e_1, \cdots, e_n\} \quad \gamma \rhd e : t \quad \exists i \in [1, n] \,.\, \gamma \rhd e \sim e_i}{\gamma \rhd e : t \textbf{ in } \{e_1, \cdots, e_n\}} \text{ [te-enum]}$$

$$\frac{\exists v \quad \gamma \rhd e_1 \rightsquigarrow v \quad \gamma \rhd e_2 \rightsquigarrow v}{\gamma \rhd e_1 \sim e_2} \text{ [te-equiv]}$$

L'équation [te-enum] exprime de manière formelle la nature même du type énuméré que nous proposons : un domaine de valuation discret et fini. Pour pouvoir contrôler si un terme e quelconque appartient à ce domaine de valuation, il doit pouvoir être calculable en un nombre fini d'étapes, et converger vers une des valeurs du domaine. De plus cette convergence doit être déterministe car sinon, la relation "e est un terme de type t **in** $\{e_1, \cdots, e_n\}$" devient elle aussi indéterministe (parfois vraie, parfois fausse) ce qui est indésirable pour raisonner sur le système de type. La propriété minimale attendue pour les termes ayant un type énuméré (mais aussi pour les éléments qui compose cette énumération) est la *confluence*, c'est à dire la terminaison du calcul et la convergence vers une valeur unique, indépendamment du "chemin" suivi lors des étapes de réduction [44]. Cette confluence permet d'établir une relation d'équivalence naturelle, formalisée par [te-equiv].

Les définitions suivantes précisent les relations entre les définitions des constantes et types de *Circus* avec le système de types.

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd e : t \quad \exists v \;\; \gamma \rhd e \rightsquigarrow v}{\gamma, x : t = v \rhd \textbf{const } x : t = e : \textbf{Unit}} \text{ [te-def]}$$

La règle [te-def] indique qu'une définition de constante est correctement typée (terme de type **Unit**) si le terme constant e est confluent vers v, si il a bien le type déclaré t et si de plus le nom qui lui est attribué, x n'est pas déjà utilisé dans le contexte de typage γ. On remarque que le contexte de typage est alors

enrichi par la nouvelle définition. C'est la notation $\gamma, x : t = v$ qui indique de plus que x, nouvelle entrée dans le contexte, est associée à son type et sa valeur normale.

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd e \preccurlyeq: t \quad \gamma \rhd e \rightsquigarrow v}{\gamma, x : t = v \rhd \textbf{const}\ x := e : \textbf{Unit}}\ [\text{te-idef}]$$

La règle [te-idef] est légèrement plus complexe, en ce sens que le type t n'est pas précisé explicitement dans la définition de la constante. Il doit être "inventé" de manière telle que t soit le type minimal de e, noté $\gamma \rhd e \preccurlyeq: t$. Cette notion de type minimal, directement liée à la relation de sous-typage sera définie formellement par la suite. À ce stade, nous retiendrons seulement que t doit correspondre étroitement à e : il serait trivial (mais peu utile) d'inférer le type $\top$ dans tous les cas où e serait bien typé.

$$\frac{\tau \notin \text{dom}(\gamma) \quad \gamma \rhd t \rightsquigarrow u}{\gamma, \tau :: u \rhd \textbf{type}\ \tau = t : \textbf{Unit}}\ [\text{te-tdef}]$$

[te-tdef] indique que les déclarations de types sont également conservées dans le contexte. De plus, les types doivent être confluents. Notons que la confluence des types n'a pas été rigoureusement définie : seule la confluence des termes a été évoquée. Nous considérerons toutefois que les types énumérés étant composés de termes, il est assez naturel de prendre en compte la notion de calcul sur les types au même niveau que le calcul sur les termes. Du reste, il convient de rappeller que les transitions autorisant le calcul des types ont été rigoureusement définies dans la section sur la sémantique opérationnelle du noyau.

$$\frac{x_3 \notin \text{dom}(\gamma) \quad x_2 \in \text{dom}(\gamma')}{\gamma, x_1 :: \gamma', x_3 : \gamma'(x_2) \rhd \textbf{from}\ x_1\ \textbf{import}\ x_2\ \textbf{as}\ x_3 : \textbf{Unit}}\ [\text{te-impdef}]$$

La règle [te-impdef] précise le typage des clauses d'importation. Elle utilise une structure "emboîtée à un seul niveau" des modules *Circus* dans lesquels les constantes et types sont définis, structure également mise en évidence dans les règles [te-mod] et [te-all-mod].

$$\frac{\gamma, \rhd d_1 : \textbf{Unit} \quad \gamma, \gamma' \rhd d_2 : \textbf{Unit}}{\gamma, \gamma' \rhd d_1\ d_2 : \textbf{Unit}}\ [\text{te-all-def}]$$

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma, \gamma' \rhd d : \textbf{Unit}}{\gamma, x :: \gamma' \rhd \textbf{module}\ x \{d\} : \textbf{Unit}}\ [\text{te-mod}]$$

$$\frac{\gamma, x :: \gamma' \rhd (\textbf{module}\ x \{d\}) : \textbf{Unit} \quad \gamma, x :: \gamma', \gamma'' \rhd\ c : \textbf{Unit}}{\gamma, x :: \gamma', \gamma'' \rhd \textbf{module}\ x \{d\}\ c : \textbf{Unit}}\ [\text{te-all-mod}]$$

4.3.4 Sous-typage

Le premier groupe de règles définit les propriétés structurelles de la relation de sous-typage : $\top$ comme type maximal, $\bot$ comme type minimal, *reflexivité* et *transitivité* de la relation, et enfin la règle [st-sub] qui permet d'utiliser la relation de sous-typage dans les dérivations, et notamment de pouvoir donner le type $\top$ à tout terme possédant un type bien formé.

$$\frac{\gamma \ \rhd \ t}{\gamma \rhd t \preccurlyeq \top} \ \text{[st-top]} \qquad \frac{\gamma \ \rhd \ t}{\gamma \rhd \bot \preccurlyeq t} \ \text{[st-bot]}$$

$$\frac{\gamma \ \rhd \ t}{\gamma \rhd t \preccurlyeq t} \ \text{[st-refl]} \qquad \frac{\gamma \rhd t_1 \preccurlyeq t_2 \quad \gamma \rhd t_2 \preccurlyeq t_3}{\gamma \rhd t_1 \preccurlyeq t_3} \ \text{[st-trans]}$$

$$\frac{\gamma \rhd e : t_1 \quad \gamma \rhd t_1 \preccurlyeq t_2}{\gamma \rhd e : t_2} \ \text{[st-sub]}$$

Notons à présent que le type **None** est uniquement sous-type des types primitifs distincts de $\bot$. De plus, les types énumérés sont supertypes de **None** uniquement s'ils contiennent la valeur **none**.

$$t \in \{\textbf{num}, \textbf{string}, \textbf{bool}, \textbf{Unit}\} \qquad \frac{}{\gamma \rhd \textbf{None} \preccurlyeq t} \ \text{[st-none]}$$

$$\frac{\gamma \rhd \textbf{none} : t \ \textbf{in} \ S}{\gamma \rhd \textbf{None} \preccurlyeq t \ \textbf{in} \ S} \ \text{[st-none2]}$$

Enfin, la description de la relation de sous-typage se termine avec les types énumérés et les types fonction. Notons qu'un type énuméré est sous-type de n'importe quel autre type à la condition que tous les termes énumérés possèdent ce même type, ce qui est une définition assez naturelle. Le sous-typage des fonctions fait classiquement apparaître une covariance sur les types des arguments de retour et une contravariance sur les types des arguments d'entrée.

$$\frac{\gamma \ \rhd \ t_1 \ \textbf{in} \ \{e_1, \cdots, e_n\} \quad \gamma \ \rhd \ t_2 \quad \gamma \rhd e_1 : t_2 \ \cdots \ \gamma \rhd e_n : t_2}{\gamma \rhd t_1 \ \textbf{in} \ \{e_1, \cdots, e_n\} \preccurlyeq t_2} \ \text{[st-enum]}$$

$$\frac{\gamma \ \rhd \ t_1 \to t_2 \quad \gamma \ \rhd \ t_3 \to t_4 \quad \gamma \rhd t_2 \preccurlyeq t_4 \quad \gamma \rhd t_3 \preccurlyeq t_1}{\gamma \rhd t_1 \to t_2 \preccurlyeq t_3 \to t_4} \ \text{[st-lamb]}$$

4.3.5 Type minimal d'un terme

De part la nature même du sous-typage proposé, un terme possède de nombreux types licites, du plus spécialisé au plus général ($\top$). Typiquement, un algorithme d'inférence de type pour un terme quelconque est intéressant si il est capable de déterminer le plus petit type licite possible (ou un des plus petits, car comme dans notre cas, ils ne sont pas forcément uniques). Afin de formaliser cette particularité, nous introduisons la notion de type minimal pour un terme e, noté $\gamma \rhd e \preccurlyeq: t$ ("e a pour type minimal t dans le contexte γ"), au moyen de la définition suivante :

Définition 4.3 *type minimal d'un terme*

$$\gamma \rhd e \preccurlyeq: t \quad \Leftrightarrow \quad \begin{cases} \gamma \rhd e : t \\ \gamma \rhd e : t' \Rightarrow \gamma \rhd t \preccurlyeq t' \end{cases} \qquad \textit{[te-min]}$$

Notons que cette relation ne définit pas un type minimal unique , mais plutôt une classe d'équivalence pour la relation d'égalité structurelle sur les types, $\doteq$, présentée dans la sous-section suivante. En effet,

les assertions suivantes sont par exemple, toutes vraies et démontrables :

$$\begin{cases} \gamma \rhd 10 \preccurlyeq: \textbf{num in } \{10\} \\ \gamma \rhd 10 \preccurlyeq: \textbf{num in } \{5+5\} \\ \gamma \rhd 10 \preccurlyeq: \top \textbf{ in } \{10\} \\ \gamma \rhd 10 \preccurlyeq: \textbf{num in } \{10, 10\} \end{cases}$$

Le théorème suivant offre une équation générale pour dériver des termes minimalement typés, lorsque ceux-ci sont confluents.

Proposition 4.4 *typage minimal d'un terme confluent*

$$\boxed{\frac{\gamma \ \rhd \ t \textbf{ in } \{e\}}{\gamma \rhd e \preccurlyeq: t \textbf{ in } \{e\}} \ [\textit{te-min2}]}$$

Preuve : En appliquant [te-min], on obtient directement

$$\frac{\gamma \rhd e : t \textbf{ in } \{e\} \qquad \gamma \rhd e : t' \Rightarrow \gamma \rhd t \textbf{ in } \{e\} \preccurlyeq t'}{\gamma \rhd e \preccurlyeq: t \textbf{ in } \{e\}}$$

La branche gauche du numérateur correspond directement à l'hypothèse et la branche droite se développe facilement en

$$\gamma \rhd e : t' \Rightarrow \frac{\gamma \ \rhd \ t \textbf{ in } \{e\} \ \ \gamma \rhd e : t'}{\gamma \rhd t \textbf{ in } \{e\} \preccurlyeq t'} [\text{st-enum}]$$

□

La notion de type minimal est intéressante à plus d'un titre. Elle permet de différencier l'utilisation de [st-sub] dans les dérivations logiques, et, dans le cas des types énumérés, de propager les informations de types sans perte, comme il le sera montré avec les opérateur $+, ==,$ **if**, **and**, **not**. En effet, il est logique d'attendre que $10 + 20$ ait comme type minimal **num in** $\{30\}$. L'équation [te-min2] permet de dériver le type minimal d'un terme littéral (et donc normal), car il est facile de montrer que dans ce cas, $\emptyset \rhd v \rightsquigarrow v$. Pour pouvoir utiliser le type minimal dans le système de type, nous redéfinissons les règles [te-env] et [te-env2] par

$$\boxed{\begin{array}{c} \gamma, x : t \rhd x \preccurlyeq: t \ \ [\text{te-env}] \\ \\ \gamma, x : t = v \rhd x \preccurlyeq: t \ \ [\text{te-env2}] \\ \\ \dfrac{\gamma \rhd e \preccurlyeq: t}{\gamma \rhd e : t} \ [\text{te-env4}] \end{array}}$$

Ces règles n'introduisent pas d'incohérence par rapport à [te-min], car le seul type dérivable pour un terme x est donné par le contexte(via [te-env]). Nous complétons à présent le système d'inférence par deux propositions pour typer minimalement les termes lambda et application.

Proposition 4.5

$$\frac{x \notin \mathit{dom}(\gamma) \quad \gamma \ \rhd \ t_1 \to t_2 \quad \gamma, x : t_1 \rhd e \preccurlyeq: t_2}{\gamma \rhd \lambda x : t_1 . e \preccurlyeq: t_1 \to t_2} \ [\textit{te-lamb-min}]$$

Preuve : En retenant les hypothèses, et notamment $e \preccurlyeq: t_2$, on montre que la définition [te-min] s'applique, car, facilement $\gamma \rhd \lambda x\!:\!t_1.e \ : \ t_1 \to t_2$, et d'autre part si il existe un type $t_1 \to t$ tel que $\gamma \rhd \lambda x\!:\!t_1.e : t_1 \to t$, alors $\gamma \rhd e : t$ ([te-lam]). Or comme $\gamma \rhd e \preccurlyeq: t_2$, alors nécessairement $\gamma \rhd t \preccurlyeq t_2$, donc d'après [st-lam], $\gamma \rhd t_1 \to t \preccurlyeq t_1 \to t_2$ □

Proposition 4.6

$$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \to t_2 \quad \gamma \rhd e_2 : t_1}{\gamma \rhd e_1(e_2) \preccurlyeq: t_2} \ [\textit{te-@-min}]$$

Preuve :similaire □

Les propositions suivantes sont les extensions naturelles des règles définissant ce typage des tests. Leur nombre vient naturellement de la combinatoire des trois valeurs possibles pouvant prendre le type **bool**. Toutefois, leur traitement et utilisation ne pose pas de problèmes car ces équations sont exclusives.

Proposition 4.7

$$\gamma \rhd \textbf{None} \preccurlyeq t \quad \frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 \preccurlyeq: t \quad \gamma \rhd e_3 \preccurlyeq: t}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t} \ [\textit{te-if-min}]$$

Preuve : similairement, en utilisant [te-if] et [te-min] □

Proposition 4.8

$$\gamma \rhd \textbf{None} \preccurlyeq t \quad \frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: t \quad \gamma \rhd e_3 \preccurlyeq: t}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t} \ [\textit{te-if-min1}]$$

Preuve : similairement, en utilisant [te-if1] et [te-min] □

Proposition 4.9

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e_2 \preccurlyeq: t \quad \gamma \rhd e_3 \preccurlyeq: t}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t} \ [\textit{te-if-min2}]$$

Preuve : similairement, en utilisant [te-if2] et [te-min] □

Proposition 4.10

$$\gamma \rhd \textbf{None} \preccurlyeq t_1 \quad \frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_1} \ [\textit{te-if-min3}]$$

Preuve : similairement, en utilisant [te-if3] et [te-min] □

Proposition 4.11

$$\gamma \rhd \textbf{None} \preccurlyeq t_2 \quad \frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 \preccurlyeq: t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_2} \ [\textit{te-if-min4}]$$

Preuve : similairement, en utilisant [te-if4] et [te-min] □

Proposition 4.12

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{true}\} \quad \gamma \rhd e_2 \preccurlyeq: t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_1} \text{ [te-if-true-min]}$$

Preuve : similairement, en utilisant [te-if-true] et [te-min] □

Proposition 4.13

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{false}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 \preccurlyeq: t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_2} \text{ [te-if-false-min]}$$

Preuve : similairement, en utilisant [te-if-false] et [te-min] □

Proposition 4.14

$$\frac{\gamma \rhd e_1 : \textbf{bool in } \{\textbf{none}\} \quad \gamma \rhd e_2 : t_1 \quad \gamma \rhd e_3 : t_2}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: \textbf{None}} \text{ [te-if-none-min]}$$

Preuve : similairement, en utilisant [te-if-none] et [te-min] □

4.3.6 Equivalences de types

La première relation $\gamma \rhd t_1 \doteq t_2$ exprime l'égalité structurelle de deux types, et offre les "bonnes" propriétés attendues pour une relation d'équivalence (reflexivité, commutativité et transitivité). De plus, elle est congruente pour les relations de termes bien typés et de termes minimalement typés, ce qui signifie que des types structurellement égaux peuvent être substitués dans les jugements de typage sans modifier leur validité. La seconde relation $\gamma \rhd t_1 \approx t_2$, moins forte, est utilisée parfois dans la suite du développement pour exprimer une certaine compatibilité entre types.

$$\frac{\gamma \rhd t_1 \preccurlyeq t_2 \quad \gamma \rhd t_2 \preccurlyeq t_1}{\gamma \rhd t_1 \doteq t_2} \text{ [t-eq]}$$

$$\frac{\gamma \rhd t_1 \preccurlyeq t_2}{\gamma \rhd t_1 \approx t_2} \text{ [t-aq1]} \qquad \frac{\gamma \rhd t_2 \preccurlyeq t_1}{\gamma \rhd t_1 \approx t_2} \text{ [t-aq2]}$$

Les propriétés suivantes résument l'essentiel de ce qui est attendu pour une égalité de types. (3) et (4) soulignent les particularités des types énumérés.

Propriété 4.15

$(1a)$	$\gamma \rhd t_1 \doteq t_1$	*reflexivité*
$(1b)$	$\gamma \rhd t_1 \doteq t_2 \Leftrightarrow \gamma \rhd t_2 \doteq t_1$	*commutativité*
$(1c)$	$\gamma \rhd t_1 \doteq t_2 \wedge \gamma \rhd t_2 \doteq t_3 \Leftrightarrow \gamma \rhd t_1 \doteq t_3$	*transitivité*
$(2a)$	*si* $\gamma \rhd t_1 \doteq t_2$ *alors* $\gamma \rhd e : t_1 \Leftrightarrow \gamma \rhd e : t_2$	$\doteq$ *congruente pour* $:$
$(2b)$	*si* $\gamma \rhd t_1 \doteq t_2$ *alors* $\gamma \rhd e \preccurlyeq: t_1 \Leftrightarrow \gamma \rhd e \preccurlyeq: t_2$	$\doteq$ *congruente pour* $\preccurlyeq:$
(3)	$\gamma \rhd t_1$ **in** $S \doteq t_2$ **in** S	
(4)	$\gamma \rhd (t$ **in** $S_1)$ **in** $S_2 \doteq t$ **in** S_2	
(5)	$\gamma \rhd$ **None** $\doteq \top$ **in** $\{$**none**$\}$	

Preuve : (1a),(1b),(1c) et (2a),(2b) sont obtenues directement en utilisant les définitions du sous-typage. Les propriétés (3), (4) et (5) utilisent l'hypothèse de confluence liée aux types énumérés bien formés. □

4.3.7 Opérations sur les types

Union minimale ou calcul du plus petit supertype commun

Nous proposons à présent un premier opérateur sur les types, noté $\oplus$ ("union minimale") qui permet d'exploiter plus finement le type énuméré, et de simplifier le calcul des types minimaux. Etant donné deux types bien formés t_1, t_2 , leur union minimale peut être vue comme un constructeur calculant le plus petit supertype commun, ce qui sera formalisé par la propriété 4.23, présentée plus loin.

Noté $\oplus$, il est défini pour tout environnement γ, par le tableau suivant :

Définition 4.16 *Union minimale de types* $\oplus$

$\oplus$	**None**	**num**	**string**	**bool**	t_2 **in** S_2	$t_{21} \to t_{22}$
None	**None**	**num**	**string**	**bool**	t_2 **in** $S_2, \{\textbf{none}\}$ [1] $\top$ *sinon*	$\top$
num	**num**	**num**	$\top$	$\top$	**num** [b] $\top$ *sinon*	$\top$
string	**string**	$\top$	**string**	$\top$	**string** [c] $\top$ *sinon*	$\top$
bool	**bool**	$\top$	$\top$	**bool**	**bool** [d] $\top$ *sinon*	$\top$
t_1 **in** S_1	t_1 **in** $S_1, \{\textbf{none}\}$ [e] $\top$ *sinon*	**num** [f] $\top$ *sinon*	**string** [g] $\top$ *sinon*	**bool** [h] $\top$ *sinon*	$\top$ **in** S_1, S_2	$t_{21} \to t_{22}$ [i] $\top$ *sinon*
$t_{11} \to t_{12}$	$\top$	$\top$	$\top$	$\top$	$t_{11} \to t_{12}$ [j] $\top$ *sinon*	$t_{11} \otimes t_{12} \to t_{12} \oplus t_{22}$

[1] si $\gamma \rhd$ **None** $\preccurlyeq t_2$
[b] si $\gamma \rhd t_2$ **in** $S_2 \preccurlyeq$ **num**
[c] si $\gamma \rhd t_2$ **in** $S_2 \preccurlyeq$ **string**
[d] si $\gamma \rhd t_2$ **in** $S_2 \preccurlyeq$ **bool**
[e] si $\gamma \rhd$ **None** $\preccurlyeq t_1$
[f] si $\gamma \rhd t_1$ **in** $S_1 \preccurlyeq$ **num**
[g] si $\gamma \rhd t_1$ **in** $S_1 \preccurlyeq$ **string**
[h] si $\gamma \rhd t_1$ **in** $S_1 \preccurlyeq$ **bool**
[i] si $\gamma \rhd t_1$ **in** $S_1 \preccurlyeq t_{21} \to t_{22}$
[j] si $\gamma \rhd t_2$ **in** $S_2 \preccurlyeq t_{11} \to t_{12}$

Notons que l'union de deux types fonction fait appel à l'opérateur d'intersection maximale, défini dans la sous-section suivante. D'autre part, pour compléter la définition, l'union d'un type t quelconque avec $\bot$ et $\top$ est définie de manière générale par :

$$t \oplus \bot \stackrel{def}{=} t \stackrel{def}{=} \bot \oplus t$$
$$\top \oplus t \stackrel{def}{=} \top \stackrel{def}{=} t \oplus \top$$

où $\stackrel{def}{=}$ signifie "égal par définition".

Intersection maximale, ou calcul du plus grand sous-type commun

L'opération notée $\otimes$, est définie pour tout environnement γ, par le tableau suivant :

Définition 4.17 *intersection de types* $\otimes$

$\otimes$	**None**	**num**	**string**	**bool**	t_2 **in** S_2	$t_{21} \rightarrow t_{22}$
None	**None**	**None**	**None**	**None**	**None**[a] $\bot$ *sinon*	$\bot$
num	**None**	**num**	**None**	**None**	t_2 **in** S_2'[b] $\bot$ *sinon*	$\bot$
string	**None**	**None**	**string**	**None**	t_2 **in** S_2'[c] $\bot$ *sinon*	$\bot$
bool	**None**	**None**	**None**	**bool**	t_2 **in** S_2'[d] $\bot$ *sinon*	$\bot$
t_1 **in** S_1	**None**[e] $\bot$ *sinon*	t_1 **in** S_1'[f] $\bot$ *sinon*	t_1 **in** S_1'[g] $\bot$ *sinon*	t_1 **in** S_1'[h] $\bot$ *sinon*	$\top$ **in** $S_1 \cap S_2$[i] $\bot$ *sinon*	t_1 **in** S_1'[j] $\bot$ *sinon*
$t_{11} \rightarrow t_{12}$	$\bot$	$\bot$	$\bot$	$\bot$	t_2 **in** S_2'[k] $\bot$ *sinon*	$t_{11} \oplus t_{21} \rightarrow t_{12} \otimes t_{22}$

[a] si $\gamma \rhd$ **None** $\preccurlyeq t_2$ **in** S_2
[b] si $S_2' \neq \emptyset$ avec $S_2' = \{e \in S_2 | \gamma \rhd e :$ **num**$\}$
[c] si $S_2' \neq \emptyset$ avec $S_2' = \{e \in S_2 | \gamma \rhd e :$ **string**$\}$
[d] si $S_2' \neq \emptyset$ avec $S_2' = \{e \in S_2 | \gamma \rhd e :$ **bool**$\}$
[e] si $\gamma \rhd$ **None** $\preccurlyeq t_1$ **in** S_1
[f] si $S_1' \neq \emptyset$ avec $S_1' = \{e \in S_1 | \gamma \rhd e :$ **num**$\}$
[g] si $S_1' \neq \emptyset$ avec $S_1' = \{e \in S_1 | \gamma \rhd e :$ **string**$\}$
[h] si $S_1' \neq \emptyset$ avec $S_1' = \{e \in S_1 | \gamma \rhd e :$ **bool**$\}$
[i] si $S_1 \cap S_2 \neq \emptyset$. *N.B : l'intersection se calcule sur des termes normalisés*
[j] si $S_1' \neq \emptyset$ avec $S_1' = \{e \in S_1 | \gamma \rhd e : t_{21} \rightarrow t_{22}\}$
[k] si $S_2' \neq \emptyset$ avec $S_2' = \{e \in S_2 | \gamma \rhd e : t_{21} \rightarrow t_{22}\}$

Notons qu'ici aussi, l'intersection de deux types fonction utilise l'opération $\oplus$ d'union de types, présentée précédemment. De plus, l'intersection d'un type quelconque t avec $\bot$ et $\top$ est définie de manière générale par

$$t \otimes \bot \stackrel{def}{=} \bot \stackrel{def}{=} \bot \otimes t$$
$$\top \otimes t \stackrel{def}{=} t \stackrel{def}{=} t \otimes \top$$

Propriétés

La première propriété fondamentale affirme que deux types bien formés donnent un type bien formé après union ou intersection.

Propriété 4.18 *union et intersection bien formées*

$$\gamma \rhd t_1, \gamma \rhd t_2 \quad \Rightarrow \quad \left\{ \begin{array}{l} \gamma \rhd t_1 \oplus t_2 \\ \gamma \rhd t_1 \otimes t_2 \end{array} \right.$$

Preuve : Par induction sur la structure des types. Les types primitifs sont éxaminés directement dans la table pour établir la propriété (non produit ici). On suppose la propriété vraie pour t_{11}, t_{12}, t_{21} et t_{22}.

cas $t_1 \equiv t_{11}$ **in** $\{e_{11} \cdots e_{1n}\}$ Les hypothèses associées sont données de manière unique par $\gamma \rhd t_1$ et $\gamma \rhd t_2$. Pour t_1 elle se développe en :

$$\frac{\gamma \rhd t_{11} \quad \gamma \rhd e_{11} : t_{11} \quad \cdots \quad \gamma \rhd e_{1n} : t_{11} \quad \gamma \rhd e_{11} \rightsquigarrow v_{11} \quad \cdots \quad \gamma \rhd e_{1n} \rightsquigarrow v_{1n}}{\gamma \rhd t_{11} \textbf{ in } \{e_{11} \cdots e_{1n}\}} \text{ [t-enum]}$$

Nous avons pour t_2 les possibilités suivantes

cas $\bot$. Par définition, $t_1 \oplus t_2 \stackrel{def}{=} t_1$ et donc $\gamma \ \rhd \ t_1$, et $t_1 \otimes t_2 \stackrel{def}{=} \bot$, et donc $\gamma \ \rhd \ \bot$

cas $\top$. Par définition, $t_1 \oplus t_2 \stackrel{def}{=} \top$ et donc $\gamma \ \rhd \ \top$, et $t_1 \otimes t_2 \stackrel{def}{=} t_1$, et $\gamma \ \rhd \ t_1$

cas None . Dans ce cas, $t_1 \oplus t_2 \stackrel{def}{=} \gamma \ \rhd \ t_{11}$ **in** $\{e_{11} \cdots e_{1n}, \textbf{none}\}$ si $\gamma \rhd \textbf{None} \preccurlyeq t_{11}$, et $\bot$ sinon. Or, pour i tel que $1 \leq i \leq n$,

$$\dfrac{\gamma \ \rhd \ t_{11} \quad \gamma \rhd e_{1i} : t_{11} \quad \gamma \rhd e_{1i} \rightsquigarrow v_{1i} \quad \dfrac{\gamma \rhd \textbf{none} : \textbf{None} \quad \gamma \rhd \textbf{None} \preccurlyeq t_{11}}{\gamma \rhd \textbf{none} : t_{11}}\,[1] \quad \mathit{conv}(\gamma, \textbf{none})}{\gamma \ \rhd \ t_{11} \textbf{ in } \{e_{11} \cdots e_{1n}, \textbf{none}\}}\,[b]$$

[1][st-sub]
[b][t-enum]

D'autre part, nous avons directement $\dfrac{}{\gamma \ \rhd \ \bot}$ [t-bot].

Pour l'intersection, soit $t_1 \otimes t_2 \stackrel{def}{=} \textbf{None}$ soit $t_1 \otimes t_2 \stackrel{def}{=} \bot$. Dans tous les cas, le résultat est bien formé.

cas $t_2 \in \{\textbf{num}, \textbf{string}, \textbf{bool}\}$ Nous avons $t_1 \oplus t_2 \stackrel{def}{=} t_2$, si $\gamma \rhd t_1 \preccurlyeq t_2$ ou $t_1 \oplus t_2 \stackrel{def}{=} \top$ sinon. Le deuxieme cas est trivial. Pour le premier, nous avons par hypothèse $\gamma \ \rhd \ t_2$. Nous avons $t_1 \otimes t_2 \stackrel{def}{=} t_{11}$ **in** $\{e_{1i}\}$ pour tout $i, 1 \leq i \leq n$ tel que $\gamma \rhd e_{1i} : t_2$. Cette définition s'applique si de tels e_{1i} existent. Dans le cas contraire le résultat est $\bot$, bien formé. Pour le premier cas, il est clair que $\gamma \ \rhd \ t_{11}$ **in** $\{e_{1i}\}$.

cas $t_2 \equiv t_{21}$ **in** $\{e_{21} \cdots e_{2k}\}$ Nous avons $t_1 \oplus t_2 \stackrel{def}{=} \top$ **in** $\{e_{11}, \cdots, e_{1n}, e_{21}, \cdots, e_{2k}\}$. Or, pour tout i, j tels que $1 \leq i \leq n$ et $1 \leq j \leq k$,

$$\dfrac{\dfrac{}{\gamma \ \rhd \ \top}\,[a] \quad \dfrac{\gamma \rhd e_{1i} : t_{11} \quad \dfrac{\gamma \ \rhd \ t_{11}}{\gamma \rhd t_{11} \preccurlyeq \top}\,[b]}{\gamma \rhd e_{1i} : \top}\,[c] \quad \dfrac{\gamma \rhd e_{2j} : t_{21} \quad \dfrac{\gamma \ \rhd \ t_{21}}{\gamma \rhd t_{21} \preccurlyeq \top}\,[d]}{\gamma \rhd e_{2j} : \top}\,[e] \quad \begin{array}{l}\gamma \rhd e_{1i} \rightsquigarrow v_{1i} \\ \gamma \rhd e_{2j} \rightsquigarrow v_{2j}\end{array}}{\gamma \ \rhd \ \top \textbf{ in } \{e_{11}, \cdots, e_{1n}, e_{21}, \cdots, e_{2k}\}}\,[f]$$

[a][t-top]
[b][st-top]
[c][st-sub]
[d][st-top]
[e][st-sub]
[f][t-enum]

D'autre part, $t_1 \otimes t_2 \stackrel{def}{=} \top$ **in** $\{e_{31} \cdots e_{3m}\}$ avec , pour tout $i \in [1, m]$, e_{3i} tels que $\gamma \rhd e_{3i} : t_1$ et $\gamma \rhd e_{3i} : t_2$. Si de tels termes n'existent pas, le résultat est $\bot$, bien formé par définition. Pour le premier résultat, il existe i tel que $1 \leq i \leq m$ et

$$\dfrac{\dfrac{}{\gamma \ \rhd \ \top}\,[a] \qquad \begin{array}{c}\exists j \in [1,n]\ \dfrac{\gamma \ \rhd \ t_{11}\ \mathbf{in}\ \{e_{11}\cdots e_{1n}\} \quad \gamma \rhd e_{3i} \sim e_{1j}}{\gamma \rhd e_{3i} : t_{11}\ \mathbf{in}\ \{e_{11}\cdots e_{1n}\}}\,[b] \\ \text{ou} \\ \exists j \in [1,k]\ \dfrac{\gamma \ \rhd \ t_{21}\ \mathbf{in}\ \{e_{21}\cdots e_{2k}\} \quad \gamma \rhd e_{3i} \sim e_{2j}}{\gamma \rhd e_{3i} : t_{21}\ \mathbf{in}\ \{e_{21}\cdots e_{2k}\}}\,[c]\end{array}}{\gamma \ \rhd \ \top\ \mathbf{in}\ \{e_{31}\cdots e_{3m}\}}\,[d]$$

[a] [t-top]
[b] [te-enum]
[c] [te-enum]
[d] [t-enum]

cas $t_2 \equiv t_{21} \to t_{22}$ Par définition, nous avons deux cas : soit $t_1 \oplus t_2 \stackrel{def}{=} t_{21} \to t_{22}$, soit $\top$ qui est bien formé par définition. Dans le premier cas,

$$\frac{\gamma \ \rhd \ t_{21} \quad \gamma \ \rhd \ t_{22}}{\gamma \ \rhd \ t_{21} \to t_{22}}\ [\text{t-lam}]$$

Pour l'intersection, nous avons soit $t_1 \otimes t_2 \stackrel{def}{=} t_{11}\ \mathbf{in}\ \{e_{11}\cdots e_{1n}\}$, soit $\bot$, bien formé par définition. Dans le premier cas, le type est bien formé par hypothèse.

cas $t_1 \equiv t_{11} \to t_{12}$ **cas** $t_2 \in \{\bot, \top, \mathbf{None}, \mathbf{num}, \mathbf{string}, \mathbf{bool}\}$. Pour l'union, le résultat est $\top$, bien formé par définition. Pour l'intersection, le résultat est $\bot$, également bien formé par définition.

cas $t_2 \equiv t_{21}\ \mathbf{in}\ \{e_{21}\cdots e_{2k}\}$ Pour l'union, nous avons comme résultat soit $t_{11} \to t_{12}$, soit $\top$, bien formé par définition. Or,

$$\frac{\gamma \ \rhd \ t_{21} \quad \gamma \ \rhd \ t_{22}}{\gamma \ \rhd \ t_{21} \to t_{22}}\ [\text{t-lam}]$$

Pour l'intersection, nous avons soit $t_{21}\ \mathbf{in}\ \{e_{21}\cdots e_{2k}\}$ bien formé par hypothèse , soit $\bot$ bien formé par définition.

cas $t_2 \equiv t_{21} \to t_{22}$ Pour l'union, nous utilisons pleinement l'hypothèse d'induction afin de prouver que le seul résultat défini est bien formé :

$$\frac{\gamma \ \rhd \ t_{11} \otimes t_{21} \quad \gamma \ \rhd \ t_{12} \oplus t_{22}}{\gamma \ \rhd \ t_{11} \otimes t_{21} \to t_{12} \oplus t_{22}}\ [\text{t-lam}]$$

Pour l'intersection, similairement,

$$\frac{\gamma \ \rhd \ t_{11} \oplus t_{21} \quad \gamma \ \rhd \ t_{12} \otimes t_{22}}{\gamma \ \rhd \ t_{11} \oplus t_{21} \to t_{12} \otimes t_{22}}\ [\text{t-lam}]$$

□

Ces opérateurs sont idempotents et commutatifs, ce qui est formalisé par :

Propriété 4.19

$$\gamma \rhd t \oplus t \doteq t$$
$$\gamma \rhd t \oplus t' \doteq t' \oplus t$$
$$\gamma \rhd t \otimes t \doteq t$$
$$\gamma \rhd t \otimes t' \doteq t' \otimes t$$

Preuve : Symétrie et diagonale des tables définissant $\oplus, \otimes$, pour tous les types primitifs. Pour les types composites, on utilise une induction sur la structure. □

Propriété 4.20

$$\gamma \rhd t_1, \gamma \rhd t_2 \Rightarrow \begin{cases} \gamma \rhd t_1 \preccurlyeq t_1 \oplus t_2 \\ \gamma \rhd t_2 \preccurlyeq t_1 \oplus t_2 \\ \gamma \rhd t_1 \otimes t_2 \preccurlyeq t_1 \\ \gamma \rhd t_1 \otimes t_2 \preccurlyeq t_2 \end{cases}$$

Preuve : Par induction sur la structure des types, et en utilisant la commutativité de $\oplus, \otimes$. □

Propriété 4.21

$$\gamma \rhd t_1 \preccurlyeq t_1' \text{ et } \gamma \rhd t_2 \preccurlyeq t_2' \quad \Rightarrow \quad \begin{cases} \gamma \rhd t_1 \oplus t_2 \preccurlyeq t_1' \oplus t_2' \\ \gamma \rhd t_1 \otimes t_2 \preccurlyeq t_1' \otimes t_2' \end{cases}$$

Preuve : Par induction sur la structure des types. □

Propriété 4.22 *Congruence de l'égalité structurelle pour $\oplus$ et $\otimes$*

$$\gamma \rhd t \doteq t' \Rightarrow \begin{cases} \gamma \rhd t \oplus t_2 \doteq t' \oplus t_2 \\ \gamma \rhd t_1 \oplus t \doteq t_1 \oplus t' \\ \gamma \rhd t \otimes t_2 \doteq t' \otimes t_2 \\ \gamma \rhd t_1 \otimes t \doteq t_1 \otimes t' \end{cases}$$

Preuve : En appliquant la propriété 4.21. □

Propriété 4.23 *Minimalité de l'union*

$$\text{Pour tout } t \text{ bien formé, } \gamma \rhd t_1 \preccurlyeq t \wedge \gamma \rhd t_2 \preccurlyeq t \quad \Rightarrow \quad \gamma \rhd t_1 \oplus t_2 \preccurlyeq t$$

Preuve : En appliquant la propriété 4.21 :

$$\gamma \rhd t_1 \preccurlyeq t \wedge \gamma \rhd t_2 \preccurlyeq t \quad \Rightarrow \quad \gamma \rhd t_1 \oplus t_2 \preccurlyeq t \oplus t$$

Puis directement, en utilisant l'idempotence (cf 4.19), la définition de l'égalité structurelle et la transitivité de la relation de sous-typage :

$$\gamma \rhd t \doteq t \oplus t \quad \Rightarrow \quad \gamma \rhd t \oplus t \preccurlyeq t \quad \Rightarrow \quad \gamma \rhd t_1 \oplus t_2 \preccurlyeq t$$

□

Propriété 4.24 *Maximalité de l'intersection*

$$\text{Pour tout } t \text{ bien formé, } \gamma \rhd t \preccurlyeq t_1 \wedge \gamma \rhd t \preccurlyeq t_2 \quad \Rightarrow \quad \gamma \rhd t \preccurlyeq t_1 \otimes t_2$$

Preuve : En appliquant la propriété 4.21 :

$$\gamma \rhd t \preccurlyeq t_1 \wedge \gamma \rhd t \preccurlyeq t_2 \quad \Rightarrow \quad \gamma \rhd t \otimes t \preccurlyeq t_1 \otimes t_2$$

Puis directement, en utilisant l'idempotence (cf 4.19), la définition de l'égalité structurelle puis la transitivité de la relation de sous-typage :

$$\gamma \rhd t \otimes t \doteq t \quad \Rightarrow \quad \gamma \rhd t \preccurlyeq t \otimes t \quad \Rightarrow \quad \gamma \rhd t \preccurlyeq t_1 \otimes t_2$$

□

Et enfin nous terminons par l'associativité pour les deux opérateurs

Propriété 4.25 *Associativité de l'union*

$$\gamma \ \rhd \ t_1, \gamma \ \rhd \ t_2, \gamma \ \rhd \ t_3 \quad \Rightarrow \quad \gamma \rhd (t_1 \oplus t_2) \oplus t_3) \doteq t_1 \oplus (t_2 \oplus t_3)$$

Preuve : Nous montrons d'abord que $\gamma \rhd (t_1 \oplus t_2) \oplus t_3 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)$, par

$\gamma \ \rhd \ t_2, \gamma \ \rhd \ t_3$	$\Rightarrow$	$\gamma \rhd t_2 \preccurlyeq t_2 \oplus t_3$	[4.20]
$\gamma \ \rhd \ t_1$	$\Rightarrow$	$\gamma \rhd t_1 \preccurlyeq t_1$	[réflexivité]
$\gamma \rhd t_1 \preccurlyeq t_1 \wedge \gamma \rhd t_2 \preccurlyeq t_2 \oplus t_3$	$\Rightarrow$	$\gamma \rhd t_1 \oplus t_2 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)$	[4.21]
$\gamma \ \rhd \ t_2, \gamma \ \rhd \ t_3$	$\Rightarrow$	$\gamma \rhd t_3 \preccurlyeq t_2 \oplus t_3$	[4.20]
$\gamma \ \rhd \ t_2, \gamma \ \rhd \ t_3$	$\Rightarrow$	$\gamma \ \rhd \ t_2 \oplus t_3$	[4.18]
$\gamma \ \rhd \ t_2 \oplus t_3, \gamma \ \rhd \ t_1$	$\Rightarrow$	$\gamma \rhd t_2 \oplus t_3 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)$	[4.20]
$\left.\begin{array}{l}\gamma \rhd t_3 \preccurlyeq t_2 \oplus t_3 \\ \gamma \rhd t_2 \oplus t_3 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)\end{array}\right\}$	$\Rightarrow$	$\gamma \rhd t_3 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)$	[transitivité]
$\left.\begin{array}{l}\gamma \rhd t_1 \oplus t_2 \preccurlyeq t_1 \oplus (t_2 \oplus t_3) \\ \gamma \rhd t_3 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)\end{array}\right\}$	$\Rightarrow$	$\gamma \rhd (t_1 \oplus t_2) \oplus t_3 \preccurlyeq t_1 \oplus (t_2 \oplus t_3)$	[4.23]

La réciproque $\gamma \rhd t_1 \oplus (t_2 \oplus t_3) \preccurlyeq (t_1 \oplus t_2) \oplus t_3$ se démontre par une méthode similaire □

Propriété 4.26 *Associativité de l'intersection*

$$\gamma \ \rhd \ t_1, \gamma \ \rhd \ t_2, \gamma \ \rhd \ t_3 \quad \Rightarrow \quad \gamma \rhd (t_1 \otimes t_2) \otimes t_3) \doteq t_1 \otimes (t_2 \otimes t_3)$$

Preuve :Même technique que pour la proposition précédante □

Nous utiliserons parfois indistinctement $t_1 \oplus t_2 \oplus t_3$ pour toute combinaison associative des trois types.

Une première application de $\oplus$

Un des premiers interêts de l'opérateur d'union de types est de nous permettre la généralisation des théorèmes initiaux servant à inférer le type minimal des termes **if** e_1 **then** e_2 **else** e_3 (en effet [teif-min] (cf. 4.7 ne permet pas de typer minimalement une expression comme **if true then** 10 **else**$'s'$). Globalement le principe est de supprimer la précondition sur le type minimal $\gamma \rhd None \preccurlyeq t$, lorsqu'elle existe, et de relâcher la contrainte d'avoir un type minimal commun pour les opérandes e_2 et e_3. Il peut sembler fastidieux d'aller si loin dans l'utilisation du type minimal de l'alternative, et pourtant, elle est d'une grande importance pour capturer des propriétés opérationnelles des spécifications . De plus de nombreux opérateurs développés ultérieurement ont une sémantique qui utilise l'alternative.

Proposition 4.27 *généralisation de [te-if-min]*

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 \preccurlyeq: t_2 \oplus \textbf{None} \quad \gamma \rhd e_3 \preccurlyeq: t_3 \oplus \textbf{None}}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_2 \oplus t_3 \oplus \textbf{None}} \ [\textit{te-if-minb}]$$

Preuve : En appliquant [te-if], on montre que, avec les hypothèses retenues sur e_2 et e_3, **if** e_1 **then** e_2 **else** $e3$ possède un type t' tel que $\gamma \rhd t_2 \oplus \textbf{None} \preccurlyeq t'$ et $\gamma \rhd t_3 \oplus \textbf{None} \preccurlyeq t'$. Pour montrer que $t_2 \oplus t_3 \oplus \textbf{None}$ est sous-type de t', on utilise l'associativité, la commutativité et les propriétés 4.21 et 4.19 :

$$\begin{array}{rcl} \gamma \rhd t_2 \oplus \textbf{None} \preccurlyeq t', \gamma \rhd t_3 \oplus \textbf{None} \preccurlyeq t' & \Rightarrow & \gamma \rhd t_2 \oplus \textbf{None} \oplus t_3 \oplus \textbf{None} \preccurlyeq t' \oplus t' \\ & \Rightarrow & \gamma \rhd t_2 \oplus t_3 \oplus \textbf{None} \oplus \textbf{None} \preccurlyeq t' \\ & \Rightarrow & \gamma \rhd t_2 \oplus t_3 \oplus \textbf{None} \preccurlyeq t' \end{array}$$

□

Proposition 4.28 *généralisation de [te-if-min1]*

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: t_2 \oplus \textbf{None} \quad \gamma \rhd e_3 \preccurlyeq: t_3 \oplus \textbf{None}}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_2 \oplus t_3 \oplus \textbf{None}} \ [\textit{te-if-min1b}]$$

Preuve : similaire, en appliquant [te-if1]

□

Proposition 4.29 *généralisation de [te-if-min2]*

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd e_3 \preccurlyeq: t_3}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_2 \oplus t_3} \ [\textit{te-if-min2b}]$$

Preuve : similaire, en appliquant [te-if2]

□

Notons par exemple que [te-if-minb] n' est pas contradictoire avec [te-if-min] (cf. 4.7, mais seulement plus générale, puisque nous avons $\gamma \rhd t \oplus t \doteq t$ par la propriété d'idempotence 4.19. Ainsi, [te-if-minb] permet d'affirmer par exemple

$$\emptyset, x : \textbf{bool} \rhd \textbf{if } x \textbf{ then } 10 \textbf{ else } \text{'s'} \preccurlyeq: \top \textbf{ in } \{10, \text{'s'}, \textbf{none}\}$$

, ce qui serait impossible avec [te-if-min] puisque 10 et 's' n'ont aucun type minimal commun. Enfin, nous proposons d'utiliser l'opérateur $\oplus$ encore une fois pour généraliser les équations [te-if-min3] et [te-if-min4] qui avaient toutes deux une condition restrictive (le type minimal doit être super-type de **None**.

Proposition 4.30 *généralisation de [te-if-min3]*

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd e_3 : t_3}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_2 \oplus \textbf{None}} \ [\textit{te-if-min3b}]$$

Preuve : en utilisant [te-if-min3] et la propriété 4.20 qui permet d'affirmer que $\gamma \rhd \textbf{None} \preccurlyeq t_2 \oplus \textbf{None}$

□

Proposition 4.31 *généralisation de [te-if-min4]*

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 : t_2 \quad \gamma \rhd e_3 \preccurlyeq: t_3}{\gamma \rhd \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \preccurlyeq: t_3 \oplus \textbf{None}} \ [\textit{te-if-min4b}]$$

Preuve : en utilisant [te-if-min4] et la propriété 4.20 qui permet d'affirmer que $\gamma \rhd$ **None** $\preccurlyeq t_3 \oplus$ **None**
□

Avec l'ensemble de règles [te-if-minb],[te-if-min1b], [te-if-min2b],[te-if-min3b],[te-if-min4b] et enfin [te-if-true-min],[te-if-false-min],[te-if-none-min], nous avons à présent un jeu complet et minimal d'équations pour inférer le type minimal des tests (la complétude est intuitivement liée au fait que tous les typages minimaux du terme booléen sont couverts).

4.3.8 Opérateurs sur les termes

Nous abordons à présent le typage des opérateurs définis par le langage.

$$\frac{\gamma \rhd e_1 : t \quad \gamma \rhd e_2 : t}{\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-eq]}$$

$$\frac{\gamma \rhd e_1 : t \quad \gamma \rhd e_2 : t \quad t \in \{\textbf{num}, \textbf{string}, \textbf{bool}\}}{\gamma \rhd e_1 \leq e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-leq]}$$

$$\frac{\gamma \rhd e_1 : \textbf{bool} \quad \gamma \rhd e_2 : \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 : \textbf{bool}} \text{ [te-and]}$$

$$\frac{\gamma \rhd e : \textbf{bool}}{\gamma \rhd \textbf{not } e : \textbf{bool}} \text{ [te-not]}$$

Notons que l'opérateur $==$ accepte des comparaisons générales, puisqu'il suffit de trouver un type commun. Toute paire de termes bien typés possède au moins le type $\top$ en commun, de part la règle [st-sub]. Ainsi, il est possible de comparer l'égalité d'une chaine de caractère et d'un nombre :

$$\frac{\dfrac{\dfrac{}{\emptyset \rhd 10 : \textbf{num}} \quad \dfrac{}{\emptyset \rhd \textbf{num} \preccurlyeq \top}}{\emptyset \rhd 10 : \top}\text{[st-sub]} \quad \dfrac{\dfrac{}{\emptyset \rhd \text{"s"} : \textbf{string}} \quad \dfrac{}{\emptyset \rhd \textbf{string} \preccurlyeq \top}}{\emptyset \rhd \text{"s"} : \top}\text{[st-sub]}}{\emptyset \rhd 10 == \text{"s"} : \textbf{bool}}$$

L'interêt de ce typage général réside dans le gain en polymorphisme. A présent, nous présentons le typage de l'opérateur $+$, ainsi que des règles autorisant des inférences sur le typage minimal des termes $e_1 + e_2$.

$$t \in \{\textbf{string}, \textbf{num}\}$$

$$\frac{\gamma \rhd e_1 : t \quad \gamma \rhd e_2 : t}{\gamma \rhd e_1 + e_2 : t} \text{ [te-plus]}$$

$$\frac{\gamma \rhd e_1 : t \textbf{ in } S_1 \quad \gamma \rhd e_2 : t \textbf{ in } S_2}{\gamma \rhd e_1 + e_2 : t \textbf{ in } S_3} \text{ [te-plus2]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t \textbf{ in } S_1 \quad \gamma \rhd e_2 \preccurlyeq: t \textbf{ in } S_2}{\gamma \rhd e_1 + e_2 \preccurlyeq: t \textbf{ in } S_3} \text{ [te-plus-min]}$$

$$S_3 \equiv \sum(S_1, S_2)$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t \textbf{ in } S_1 \quad \gamma \rhd e_2 \preccurlyeq: t}{\gamma \rhd e_1 + e_2 \preccurlyeq: t} \text{ [te-plus-min2]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t \quad \gamma \rhd e_2 \preccurlyeq: t \textbf{ in } S_2}{\gamma \rhd e_1 + e_2 \preccurlyeq: t} \text{ [te-plus-min3]}$$

Dans [te-plus-min], la notation $\sum(S_1, S_2)$ est définie par

$$\sum(\{e_1, \cdots, e_n\}, \{e'_1, \cdots, e'_k\}) \quad \equiv \quad \{e_1 + e'_1, \cdots, e_n + e'_1, \cdots, e_n + e'_k\}$$

Ces équations permettent de propager les types énumérés lors des opérations d'addition sur les termes, et donc de bénéficier de l'information extensive sur le domaine de valuation des variables. Pour clore cette caractérisation du typage minimal, nous proposons de compléter le système d'inférence avec des thèorèmes qui tirent pleinement partie des connaissances calculatoires apportées par nos types énumérés.

Proposition 4.32

$$\forall\, i \in [1, n], \forall\, j \in [1, k]$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \textbf{ in } \{e_1, \cdots, e_n\} \quad \gamma \rhd e_2 \preccurlyeq: t_2 \textbf{ in } \{e'_1, \cdots, e'_k\} \quad \neg(\gamma \rhd e_i \sim e'_j)}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{false}\}} \text{ [*te-eq-false*]}$$

Preuve : En démontrant, à partir des transitions de la sémantique opérationnelle et de [te-min2] que

$$\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{e_1 == e_2\}$$

Ensuite, on peut montrer que

$$\gamma \rhd \textbf{bool in } \{e_1 == e_2\} \doteq \textbf{bool in } \{\textbf{false}\}$$

et donc, comme l'égalité sur les types est congruente (cf propriété 4.15, alinéa (2b)),

$$\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{false}\}$$

□

Notons qu'il serait tentant d'introduire un théorème plus général que [te-eq-false], dont le numérateur serait :

$$\gamma \rhd e_1 : t_1 \quad \gamma \rhd e_2 : t_2 \quad \neg(\gamma \rhd t_1 \approx t_2)$$

Pourtant il est impossible à établir à cause de la valeur **none**, qui possède (presque) tous les types : **none** $==$ **none** converge vers la valeur **true**, de part la définition de la sémantique opérationnelle (cf règle [e-eq-none]).

Proposition 4.33

$$\frac{\gamma \rhd e_1 : t_1 \textbf{ in } \{e\} \quad \gamma \rhd e_2 : t_2 \textbf{ in } \{e'\} \quad \gamma \rhd e \sim e'}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}\}} \ [\textit{te-eq-true}]$$

Preuve :identique à la proposition précédente □

On peut lui associer la proposition suivante dont la démonstration se fait selon un schéma identique :

Proposition 4.34

$$\frac{\gamma \rhd e_1 : \textbf{None} \quad \gamma \rhd e_2 : \textbf{None}}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}\}} \ [\textit{te-eq-true2}]$$

Proposition 4.35 $t \notin \{t_1 \textbf{ in } S_1, \bot\}$, t' *quelconque*

$$\frac{\gamma \rhd e_1 \preccurlyeq: t \quad \gamma \rhd e_2 \preccurlyeq: t' \textbf{ in } S}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \ [\textit{te-eq-min1}]$$

Preuve : Voir l'annexe. □

Enfin, pour completer le typage minimal des expressions $e_1 == e_2$, il nous reste à proposer les théorèmes suivants (qui sont tous incompatibles, garantissant que les termes $e_1 == e_2$ ne peuvent avoir deux types minimaux distincts, par rapport à la relation $\doteq$)

Proposition 4.36 $t_1, t_2 \not\equiv t \textbf{ in } S$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd t_1 \approx t_2}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \ [\textit{te-eq-min2a}]$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \textbf{ in } S_1 \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd t_1 \approx t_2}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \ [\textit{te-eq-min2b}]$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \textbf{ in } S_1 \quad \gamma \rhd e_2 \preccurlyeq: t_2 \textbf{ in } S_2 \quad \gamma \rhd t_1 \textbf{ in } S_1 \approx t_2 \textbf{ in } S_2}{\gamma \rhd e_1 == e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \ [\textit{te-eq-min2c}]$$

Nous complétons le typage minimal des termes booléens par trois théorèmes de base, qui seront développés imédiatement ensuite.

Proposition 4.37 *typage minimal des termes* **not** e

$$\frac{\gamma \rhd e \preccurlyeq: \textbf{bool in } \{e_1, \cdots, e_n\}}{\gamma \rhd \textbf{not } e \preccurlyeq: \textbf{bool in } \{\textbf{not } e_1, \cdots, \textbf{not } e_n\}} \ [\textit{te-not1}] \qquad \frac{\gamma \rhd e \preccurlyeq: \textbf{bool}}{\gamma \rhd \textbf{not } e \preccurlyeq: \textbf{bool}} \ [\textit{te-not2}]$$

Preuve : utilise la définition [te-min] □

Proposition 4.38 *typage minimal des termes* e_1 **and** e_2

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{e_1, \cdots, e_n\} \quad \gamma \rhd e_2 : \textbf{bool in } \{e'_1, \cdots, e'_k\}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool in } S_3} \ [\textit{te-and1}]$$

avec

$$S_3 \equiv \{e_1 \textbf{ and } e'_1, \cdots, e_1 \textbf{ and } e'_k, \cdots, e_n \textbf{ and } e'_k\}$$

Preuve : utilise la définition [te-min] □

Afin de "capturer" le typage des termes booléens comprenant le type minimal **bool**, nous proposons d'utiliser [te-not1], [te-not2] et [te-and1] à l'aide d'un nouvel axiome, purement simplificateur, qui exprime l'équivalence "sémantique" entre les types **bool** et **bool in** $\{$**true**, **false**, **none**$\}$ [1].

Axiome 4.39

$$\frac{\gamma \rhd e \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\}}{\gamma \rhd e \preccurlyeq: \textbf{bool}} \text{ [}\textit{te-min-bool}\text{]}$$

Notons que cette implication ne peut pas être prouvée dans notre système d'inférence, car aucune règle ne permet d'établir :

$$\gamma \rhd \textbf{bool} \preccurlyeq \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\}$$

L'ensemble des théorèmes proposés ci-dessous utilise cet axiome afin de compléter la panoplie d'inférence pour les expressions booléennes basées sur **and**. En effet, bien que [te-and1] soit très générale, elle ne permet pas d'inférer les types minimaux de termes **and** lorsque une des opérandes possède le type minimal **bool**.

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 : \textbf{bool in } \{\textbf{false}\}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool in } \{\textbf{false}, \textbf{none}\}} \text{ [te-and2a]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 : \textbf{bool in } \{\textbf{none}\}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool in } \{\textbf{false}, \textbf{none}\}} \text{ [te-and2b]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 : \textbf{bool in } \{\textbf{true}\}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool}} \text{ [te-and2c]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 : \textbf{bool in } \{\textbf{false}, \textbf{true}\}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool}} \text{ [te-and2d]}$$

[1] Correspond au fait que **bool** est un type à domaine de valuation fini

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool}} \text{ [te-and3a]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{false}\} \quad \gamma \rhd e_2 : \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool in } \{\textbf{false}\}} \text{ [te-and3b]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool in } \{\textbf{none}\}} \text{ [te-and3c]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool}} \text{ [te-and3d]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool}} \text{ [te-and3e]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool in } \{\textbf{false}, \textbf{none}\} \quad \gamma \rhd e_2 : \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool in } \{\textbf{false}, \textbf{none}\}} \text{ [te-and3f]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{bool} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{bool}}{\gamma \rhd e_1 \textbf{ and } e_2 \preccurlyeq: \textbf{bool}} \text{ [te-and4]}$$

4.4 Correction du système de types

Cette section se propose de caractériser le système de type, et plus précisément, d'établir qu'il est correct par rapport à la sémantique opérationnelle, en ce sens qu'un terme bien typé ne peut pas produire une erreur d'évaluation (plus précisément, les transitions successives ne peuvent conduire à la valeur distinguée **error**). Nous procédons en plusieurs étapes : définition de la notion d'environnement d'exécution correctement typé, mais également en conformité avec un contexte de typage donné. Ensuite, nous montrons que tout terme, indépendamment de l'existence d'un type correct ou non, est calculable dans tout contexte correct. Le résultat est soit un autre terme, soit la valeur distinguée **error**. Nous complétons ensuite cette caractérisation par un théorème central sur la préservation du type lors des dérivations (qui est l'essence du critère de correction "opérationnel") puis enfin un dernier théorème qui montre que le calcul des types préserve leur structure, en ce sens qu'un type normalisé est structurellement égal au type avant sa réduction (plus exactement, chaque transition atomique préserve la structure du type).

4.4.1 Environnement d'exécution correctement typé et conforme

La structure d'exécution $\mathcal{S}$ contient les constantes et variables définies dans les expressions *Circus*. Ce sont des termes ou types "normaux", donc fermés. Cette structure est correctement typée, notée $\emptyset \rhd \mathcal{S}$, lorsque tous ses termes sont bien typés, et tous ses types sont bien formés.

Définition 4.40 *Environnement d'exécution correctement typé ($\emptyset \rhd \mathcal{S}$)*

$$\emptyset \rhd \mathcal{S} \quad \textit{ssi} \quad \left\{ \begin{array}{lll} x = v \in \mathcal{S} & \Rightarrow & \textit{il existe } t \textit{ tel que} \quad \emptyset \rhd v : t \\ \tau = u \in \mathcal{S} & \Rightarrow & \emptyset \rhd u \\ x = \mathcal{S}' \in \mathcal{S} & \Rightarrow & \emptyset \rhd \mathcal{S}' \end{array} \right.$$

A présent, nous proposons une relation de conformité entre des environnements d'exécution et un contexte de typage.

Définition 4.41 *Environnement d'exécution conforme à γ*

$$\Vdash \mathcal{S} : \gamma \qquad \textit{ssi} \left\{ \begin{array}{lll} x = v \in \mathcal{S}, & \Rightarrow & \textit{il existe } t \textit{ tel que} \quad \gamma \rhd x : t \textit{ et } \gamma \rhd v : t \\ \tau = u \in \mathcal{S}, & \Rightarrow & \tau :: t' \in \gamma \textit{ et } \gamma \rhd u \doteq t' \\ x = \mathcal{S}' \in \mathcal{S}, & \Rightarrow & \textit{il existe } \gamma' \textit{ tel que} \quad x :: \gamma' \in \gamma \textit{ et } \Vdash \mathcal{S}' : \gamma' \end{array} \right.$$

La propriété suivante découle naturellement des définitions proposées

Corollaire 4.42

$$\Vdash \mathcal{S} : \gamma \Rightarrow \emptyset \rhd \mathcal{S}$$

4.4.2 Complétude du système de transition

Le théorème suivant a pour utilité de caractériser la sémantique opérationnelle. Il permet d'affirmer que quelque soit le terme e défini par la syntaxe du langage, son évaluation produit un autre terme du langage, ou bien conduit à une erreur. Notons qu'aucune correspondance n'est établie entre les termes bien typés et leur évaluation sans erreur d'exécution. L'interêt de ce théorème est de prouver que le système de transition est complet par rapport à l'ensemble des termes engendrés par la grammaire : aucun terme ne peut être bloqué en cours d'évaluation par absence d'une équation décrivant son évolution.

Proposition 4.43 *complétude du sytème de transition par rapport à la syntaxe*

$$\textit{Pour tout terme e et tout environnement bien formé } \mathcal{S} \begin{cases} \textit{soit } [\mathcal{S} \vdash e] \to [\mathcal{S}' \vdash e'] \\ \textit{soit } [\mathcal{S} \vdash e] \to [\mathcal{S} \vdash \textbf{error}] \end{cases}$$

Preuve : Par induction sur la structure des termes. □

4.4.3 Préservation du type lors des réductions

Nous proposons un théorème établissant la correction des types pour les expressions e du langage. Il établit la correction forte du système de type par rapport à l'exécution (une expression bien typée ne peut pas conduire à une erreur d'exécution liée au type), car une erreur rencontrée lors de l'évaluation (valeur distinguée **error**) ne peut pas être typée.

Proposition 4.44 *Préservation du type lors des réductions*

$$\gamma \rhd e : t \quad \Rightarrow \quad \begin{cases} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma, \ [\mathcal{S} \vdash e] \to [\mathcal{S}' \vdash e'] \\ \textit{avec } \ \gamma \rhd e' : t \ \textit{ et } \ \Vdash \mathcal{S}' : \gamma \end{cases}$$

Preuve : En annexe □

4.4.4 Préservation structurelle d'un type lors de ses dérivations

La sémantique de la réduction des types, définie dans le système de transition, garantit que lorsque un type est réduit, celui-ci conserve ses "propriétés" par rapport au système de type (la relation $\doteq$ est congruente, cf. 4.15)

Propriété 4.45 *Égalité structurelle d'un type et de ses dérivations*

$$\gamma \ \rhd \ t \quad \Rightarrow \quad \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma, [\mathcal{S} \vdash t] \to [\mathcal{S} \vdash t'] \textit{ et } \gamma \rhd t \doteq t'$$

Preuve : Par induction sur la structure des types.

Cas où t $\in \{\textbf{num}, \textbf{bool}, \textbf{string}, \top, \bot, \textbf{Unit}\}$**.** Tous ces types vérifient $\gamma \ \rhd \ t$, et la règle [e-tbase] montre qu'ils sont sous forme normale ; donc $[\mathcal{S} \vdash t] \to [\mathcal{S} \vdash t]$ et $\gamma \rhd t \doteq t$.

Cas $t_1 \to t_2$**.** Nous avons comme unique règle pout inférer $\gamma \ \rhd \ t_1 \to t_2$:

$$\frac{\gamma \ \rhd \ t1 \quad \gamma \ \rhd \ t_2}{\gamma \ \rhd \ t_1 \to t_2} \ \text{[t-lam]}$$

On peut donc considérer que $\gamma \ \rhd \ t_1$ et $\gamma \ \rhd \ t_2$ sont impliqués par notre hypothèse.

Si t_1 **n'est pas normal** Alors [e-tlam1] nous permet d'affirmer

$$\frac{[\mathcal{S} \vdash t_1] \to [\mathcal{S} \vdash t_1']}{[\mathcal{S} \vdash t_1 \to t_2] \to [\mathcal{S} \vdash t_1' \to t_2]} \ \text{[e-tlam1]}$$

Or de par l' hypothèse d'induction,

$$\gamma \ \rhd \ t_1 \quad \Rightarrow \quad \forall \mathcal{S}, \Vdash \mathcal{S} : \gamma, [\mathcal{S} \vdash t_1] \to [\mathcal{S} \vdash t_1'] \text{ et } \gamma \rhd t_1 \doteq t_1'$$

De part [st-lam], nous avons

$$\frac{\gamma \ \rhd \ t_1 \to t_2 \ \ \gamma \ \rhd \ t_1' \to t_2 \ \ \gamma \rhd t_2 \preccurlyeq t_2 \ \ \gamma \rhd t_1' \preccurlyeq t_1}{\gamma \rhd t_1 \to t_2 \preccurlyeq t_1' \to t_2}[\text{st-lam}]$$

et également

$$\frac{\gamma \ \rhd \ t_1 \to t_2 \ \ \gamma \ \rhd \ t_1' \to t_2 \ \ \gamma \rhd t_2 \preccurlyeq t_2 \ \ \gamma \rhd t_1 \preccurlyeq t_1'}{\gamma \rhd t_1' \to t_2 \preccurlyeq t_1 \to t_2}[\text{st-lam}]$$

Tous les numérateurs sont prouvables sans difficultés à l'aide des hypothèses que nous avons développées. Nous arrivons donc sur la conclusion,

$$\frac{\gamma \rhd t_1 \to t_2 \preccurlyeq t_1' \to t_2 \ \ \gamma \rhd t_1' \to t_2 \preccurlyeq t_1 \to t_2}{\gamma \rhd t_1 \to t_2 \doteq t_1' \to t_2} \ [\text{t-eq}]$$

Si t_2 n'est pas normal Alors [e-tlam2] décrit la réduction du type $t_1 \to t_2$, et la preuve est identique au cas précédent

Cas t_1 in $\{e_1, \cdots, e_n\}$. **Si t_1 n'est pas normal.** De part [e-tenum] et l'hypothèse d'induction, il peut être montré de manière similaire par [st-enum] que

$$\gamma \rhd t_1 \textbf{ in } \{e_1, \cdots, e_n\} \preccurlyeq t_1' \textbf{ in } \{e_1, \cdots, e_n\}$$

et

$$\gamma \rhd t_1' \textbf{ in } \{e_1, \cdots, e_n\} \preccurlyeq t_1 \textbf{ in } \{e_1, \cdots, e_n\}$$

et donc qu'il sont structurellement égaux, par [t-eq].

Si t_1 est normal (noté u_1) La règle [e-tenum2] nous indique que les termes e_i sont réduits un à un, dans l'ordre d'occurence. L'hypothèse $\gamma \ \rhd \ u_1$ **in** $\{e_1, \cdots, e_n\}$ se ramène par [t-enum] à

$$i \in [1, n], \gamma \ \rhd \ u_1, \gamma \rhd e_i : u_1, \mathit{conv}(\gamma, e_i)$$

Or de part la propriété de préservation du type dans les réductions 4.44,

$$\gamma \rhd e_i : u_1 \Rightarrow \gamma \rhd e_i' : u_1$$

Et de part les propriétés de convergence [t-conv] et [te-conv]

$$\gamma \rhd e_i : \top \wedge \mathit{conv}(\gamma, e_i) \Rightarrow \gamma \rhd e_i \rightsquigarrow v \Rightarrow \gamma \rhd e_i' \rightsquigarrow v$$

Nous avons donc par [st-enum] ($t' \equiv u_1$ **in** $\{v_1, \cdots, e_i', \cdots, e_n\}$)

$$\frac{\gamma \ \rhd \ u_1 \textbf{ in } \{v_1, \cdots, e_i, \cdots, e_n\} \ \ \gamma \ \rhd \ t' \ \ \gamma \rhd v_1 : t' \ \ \cdots \gamma \rhd e_i : t' \ \ \cdots \gamma \rhd e_n : t'}{\gamma \rhd u_1 \textbf{ in } \{v_1, \cdots, e_i, \cdots, e_n\} \preccurlyeq t'}$$

En effet, la première hypothèse du numérateur est évidente (dans les hypothèse de travail). La seconde est une conséquence de la proposition sur la préservation des types 4.44 (cf [t-enum]). Les termes $v_k, \quad k \in [1, i-1]$ sont sous forme normale, et donc $\gamma \rhd v_k \rightsquigarrow v_k$ de

part [te-conv], et par conséquent $\gamma \rhd v_k \sim v_k$ par [te-equiv]. Les termes $e_k, \quad k \in [i+1, n]$ ne sont pas normaux, mais verifient $\gamma \rhd e_k \sim e_k$ ([te-equiv], [te-conv]). Il reste à prouver $\gamma \rhd e_i \sim e'_i$ par :

$$\dfrac{\exists v \;\; \gamma \rhd e_i \rightsquigarrow v \quad \dfrac{\dfrac{\gamma \rhd e_i \rightsquigarrow v}{\Vdash \mathcal{S} : \gamma \;\; [\mathcal{S} \vdash e_i] \to [\mathcal{S} \vdash e'_i] \;\; \gamma \rhd e'_i \rightsquigarrow v}[\text{te-conv}]}{\gamma \rhd e'_i \rightsquigarrow v}}{\gamma \rhd e_i \sim e'_i}\,[\text{te-equiv}]$$

□

4.4.5 Correction globale du système de types

Proposition 4.46 *Correction des déclarations*

si $\gamma \rhd d$: **Unit**, $\Vdash \mathcal{S} : \gamma$ *et* $[\mathcal{S} \vdash d] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{unit}]$ *alors* $\Vdash \mathcal{S}' : \gamma$

Preuve : En raisonnant sur la structure des termes d et en utilisant le théorème 4.44 sur la préservation du type lors des réductions, associé aux hypothèses de confluence présentes dans [te-def], [te-idef] et [te-tdef]. □

Proposition 4.47 *Correction globale du système de types*

si $\gamma \rhd c$: **Unit**, $\Vdash \mathcal{S} : \gamma$ *et* $[\mathcal{S} \vdash c] \twoheadrightarrow [\mathcal{S}' \vdash \textbf{unit}]$ *alors* $\Vdash \mathcal{S}' : \gamma$

Preuve : En raisonnant sur la structure des termes c, et en utilisant les théorèmes 4.44 et 4.46 □

4.5 Interprétation

Une sémantique opérationnelle structurée ne conduit pas forcément à un algorithme d'évaluation pertinent et réaliste. Il est donc intéressant *per se* d'étudier une fonction d'interprétation et son rapport avec le système de transition. D'autre part, de part la nature même des types énumérés, le contrôle de type réclame parfois la réduction des termes qui constituent le domaine licite de valuation. Dans notre cas il est donc nécéssaire d'intégrer à notre étude les algorithmes d'évaluation, et de les caractériser par rapport au système de transition. Nous présentons l'interpréteur au moyen de quatre algorithmes qui permettent d'évaluer les termes du langage *Circus* en respectant la sémantique opérationnelle définie par le système de transition $\rightarrow \subset (\mathcal{S} \times \boldsymbol{e}) \times (\mathcal{S} \times (\boldsymbol{e} \cup \{\textbf{error}\}))$. Le premier algorithme calcule les transitions atomiques. L'évaluation complète d'un terme e dans un contexte $\mathcal{S}$ requiert donc une succession de calculs atomiques, en propageant correctement le contexte et le terme transformé, c'est ce que réalise le second algorithme proposé. Le troisième calcule les types, et utilise lui-même le calcul des termes. Enfin le dernier algorithme calcule les définitions, constantes ou types, déclarées dans le corps des modules *Circus*.

Notons qu'à ce niveau de la présentation, *Circus* est déterministe, et même fini, en ce sens que seuls des calculs finis peuvent être spécifiés (pas de fonctions récursives, ni d'itérateurs). Cependant, ces deux caractéristiques seront perdues lors de l'extension du langage, avec l'introduction de primitives de filtrage et de parallélisme d'une part, et avec l'introduction d'un itérateur d'autre part.

4.5.1 Calcul des dérivations élémentaires

Definition

Nous introduisons d'abord une fonction **Normal**$(\,)$ qui décide si un terme e est normal. Cette dernière évoluera en complexité lors de l'introduction de nouveaux types de données.

Définition 4.48 *algorithme de décision sur la normalité des termes*

$$\begin{array}{lcl} \textbf{Normal}(\,) & : & \boldsymbol{e} \rightarrow \{\textbf{true}, \textbf{false}\} \\ \\ \textbf{Normal}(e) & = & \textit{if } e \in \{\textbf{n}, \textbf{s}, \textbf{true}, \textbf{false}, \textbf{none}, \textbf{unit}, \lambda X : t.e'\} \textit{ then return } \textbf{true} \\ & & \textit{else return } \textbf{false} \end{array}$$

L'algorithme de calcul élémentaire des termes se présente sous la forme d'une fonction dont la signature est :

$$\mathcal{E}val : \mathcal{S} \times \boldsymbol{e} \rightarrow \mathcal{S} \times (\boldsymbol{e} \cup \{\textbf{error}\})$$

Définition 4.49 *algorithme de calcul des transitions atomiques*

$\mathcal{E}val(\mathcal{S}, e)$ =
if **Normal**(e) *then return* $(\mathcal{S}, e)$
elif $(e \in \{\mathbf{x}, \tau\})$ *then*
if $e \in$ *dom*$(\mathcal{S})$ *then return* $(\mathcal{S}, \mathcal{S}(e))$
else return $(\mathcal{S}, \mathbf{error})$
elif $(e \equiv e_1$ **and** $e_2)$ *then*
if $e_1 \equiv$ **true** *then return* $(\mathcal{S}, e_2)$
elif $e_1 \equiv$ **false** *then return* $(\mathcal{S}, e_1)$
elif $e_1 \equiv$ **none** *then return* $(\mathcal{S}, \mathbf{none})$
elif $e_1 \equiv$ **error** *then return* $(\mathcal{S}, e_1)$
elif **Normal**(e_1) *then return* $(\mathcal{S}, \mathbf{error})$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_1)$
return $(s, e'$ **and** $e_2)$;
elif $(e \equiv$ **not** $e_1)$ *then*
if $e_1 \equiv$ **true** *then return* $(\mathcal{S}, \mathbf{false})$
elif $e_1 \equiv$ **false** *then return* $(\mathcal{S}, \mathbf{true})$
elif $e_1 \equiv$ **none** *then return* $(\mathcal{S}, \mathbf{none})$
elif $e_1 \equiv$ **error** *then return* $(\mathcal{S}, e_1)$
elif **Normal**(e_1) *then return* $(\mathcal{S}, \mathbf{error})$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_1)$
return $(s,$ **not** $e')$

$\mathcal{E}val(\mathcal{S}, e)$ =
elif $(e \equiv e_1 + e_2)$ *then*
if $e_1 \equiv n_1$ *then*
if $e_2 \equiv n_2$ *then return* $(\mathcal{S}, \epsilon^{-1}(\epsilon(n_1) + \epsilon(n_2)))$
elif $e_2 \equiv$ **error** *then return* $(\mathcal{S}, e_2)$
elif **Normal**(e_2) *then return* $(\mathcal{S}, \mathbf{error})$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_2)$
return $(s, n_1 + e')$
elif $e_1 \equiv s_1$ *then*
if $e_2 \equiv s_2$ *then return* $(\mathcal{S}, s_1 s_2)$
elif $e_2 \equiv$ **error** *then return* $(\mathcal{S}, e_2)$
elif **Normal**(e_2) *then return* $(\mathcal{S}, \mathbf{error})$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_2)$
return $(s, s_1 + e')$
elif $e_1 \equiv$ **error** *then return* $(\mathcal{S}, e_1)$
elif **Normal**(e_1) *then return* $(\mathcal{S}, \mathbf{error})$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_1)$
return $(s, e' + e_2)$

$\mathcal{E}val(\mathcal{S}, e)$ =

elif $(e \equiv e_1 \leq e_2)$ *then*
if $e_1 \equiv n_1$ *then*
if $e_2 \equiv n_2$ *then return* $(\mathcal{S}, \epsilon^{-1}(\epsilon(n_1) \leq \epsilon(n_2)))$
elif $e_2 \equiv$ **error** *then return* $(\mathcal{S}, e_2)$
elif **Normal**(e_2) *then return* $(\mathcal{S},$ **error**$)$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_2)$
return $(s, n_1 \leq e')$
elif $e_1 \equiv s_1$ *then*
if $e_2 \equiv s_2$ *then return* $(\mathcal{S}, s_1 == s_2[0 : len(s1)])$
elif $e_2 \equiv$ **error** *then return* $(\mathcal{S}, e_2)$
elif **Normal**(e_2) *then return* $(\mathcal{S},$ **error**$)$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_2)$
return $(s, s_1 \leq e')$
elif $e_1 \equiv$ **none** *then return* $(\mathcal{S},$ **true**$)$
elif $e_2 \equiv$ **none** *then return* $(\mathcal{S},$ **false**$)$
elif $e_1 \equiv$ **error** *then return* $(\mathcal{S}, e_1)$
elif **Normal**(e_1) *then return* $(\mathcal{S},$ **error**$)$
else
$s, e' = \mathcal{E}val(\mathcal{S}, e_1)$
return $(s, e' \leq e_2)$

$$
\begin{array}{lll}
\mathcal{E}val(\mathcal{S},e) & = & \\
 & & \textit{elif } (e \equiv e_1 == e_2) \textit{ then} \\
 & & \quad \textit{if } \textbf{Normal}(e_1) \textit{ then} \\
 & & \quad\quad \textit{if } \textbf{Normal}(e_2) \textit{ then return } (\mathcal{S}, e_1 \equiv e_2) \\
 & & \quad\quad \textit{else} \\
 & & \quad\quad\quad s, e' = \mathcal{E}val(\mathcal{S}, e_2) \\
 & & \quad\quad\quad \textit{return } (s, e_1 == e') \\
 & & \quad \textit{else} \\
 & & \quad\quad\quad s, e' = \mathcal{E}val(\mathcal{S}, e_1) \\
 & & \quad\quad\quad \textit{return } (s, e' == e_2) \\
 & & \textit{elif } (e \equiv \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \textit{ then} \\
 & & \quad \textit{if } (e_1 \equiv \textbf{true}) \textit{ then return } (\mathcal{S}, e_1) \\
 & & \quad \textit{elif } (e_1 \equiv \textbf{false}) \textit{ then return } (\mathcal{S}, e_2) \\
 & & \quad \textit{elif } (e_1 \equiv \textbf{none}) \textit{ then return } (\mathcal{S}, \textbf{none}) \\
 & & \quad \textit{elif } \textbf{Normal}(e_1) \textit{ then return } (\mathcal{S}, \textbf{error}) \\
 & & \quad \textit{else} \\
 & & \quad\quad\quad s, e' = \mathcal{E}val(\mathcal{S}, e_1) \\
 & & \quad\quad\quad \textit{return } (s, \textbf{if } e' \textbf{ then } e_2 \textbf{ else } e_3) \\
 & & \textit{elif } (e \equiv e_1(e_2)) \textit{ then} \\
 & & \quad \textit{if } e_1 \equiv \lambda x : t.e_3 \textit{ then} \\
 & & \quad\quad \textit{if } e_2 \equiv \textbf{error} \textit{ then return } (\mathcal{S}, e_2) \\
 & & \quad\quad \textit{elif } \textbf{Normal}(e_2) \textit{ then return } (\mathcal{S}, e_3[e_2/x]) \\
 & & \quad\quad \textit{else} \\
 & & \quad\quad\quad\quad s, e' = \mathcal{E}val(\mathcal{S}, e_2) \\
 & & \quad\quad\quad\quad \textit{return } (s, e_1(e')) \\
 & & \quad \textit{if } e_1 \equiv \textbf{error} \textit{ then return } (\mathcal{S}, \textbf{error}) \\
 & & \quad \textit{elif } \textbf{Normal}(e_1) \textit{ then return } (\mathcal{S}, \textbf{error}) \\
 & & \quad \textit{else} \\
 & & \quad\quad\quad s, e' = \mathcal{E}val(\mathcal{S}, e_1) \\
 & & \quad\quad\quad \textit{return } (s, e'(e_2)) \\
 & & \textit{else return } \textbf{error}
\end{array}
$$

4.5.2 Calcul de Fermeture

Définition 4.50 *algorithme de calcul de fermeture*

$$
\begin{array}{lll}
\mathcal{E}valAll & : & \mathcal{S} \times \boldsymbol{e} \to \mathcal{S} \times (\boldsymbol{e} \cup \{\textbf{error}\}) \\
 & & \\
\mathcal{E}valAll(\mathcal{S},e) & = & \textbf{while } (e \not\equiv \textbf{error}) \textbf{ and } (\textbf{Normal}(e) == \textbf{false}) \textbf{ do} \\
 & & \quad (\mathcal{S}, e) = \mathcal{E}val(\mathcal{S}, e) \\
 & & \textbf{done} \\
 & & \textit{return } (\mathcal{S}, e)
\end{array}
$$

4.5.3 Evaluation des types

Définition 4.51 *algorithme de décision sur la normalité des types (***NormalType** $:\ t \to \{$**true**, **false**$\}$*)*

$$
\begin{array}{rcl}
\textbf{NormalType}(t) & = & \textit{if } t \in \{\textbf{num}, \textbf{string}, \textbf{bool}, \textbf{None}, \top, \bot\} \textit{ then return } \textbf{true} \\
& & \textit{elif } t \equiv t_1 \to t_2 \textit{ then} \\
& & \quad \textit{if } \textbf{NormalType}(t_1) \textbf{ and } \textbf{NormalType}(t_2) \textit{return } \textbf{true} \\
& & \quad \textit{else return } \textbf{false} \\
& & \textit{elif } t \equiv t_1 \textbf{ in } \{e_1, \cdots, e_n\} \textit{ then} \\
& & \quad \textit{if } \textbf{NormalType}(t_1) \textit{ then} \\
& & \quad \textbf{for } e \in e_1 \cdots e_n \textbf{ do} \\
& & \quad\quad \textit{if } \textbf{Normal}(e) == \textbf{false } \textit{then return } \textbf{false} \\
& & \quad \textbf{done} \\
& & \quad \textit{return } \textbf{true}
\end{array}
$$

Définition 4.52 *algorithme de calcul des types (*$\mathcal{E}valType\ :\ \mathcal{S} \times \boldsymbol{t} \to \mathcal{S} \times (\boldsymbol{t} \cup \{$**error**$\})$*)*

$$
\begin{array}{rcl}
\mathcal{E}valType(\mathcal{S}, t) & = & \textit{if } \textbf{NormalType}(t) \textit{ then return } (\mathcal{S}, t) \\
& & \textit{elif } t \equiv \tau \textit{ then return } \mathcal{E}valType(\mathcal{S}, \mathcal{S}(t)) \\
& & \textit{elif } t \equiv t_1 \to t_2 \textit{ then} \\
& & \quad \textit{if } \textbf{NormalType}(t_1) \textit{ then} \\
& & \quad\quad \textit{if } t_2 \equiv \textbf{error } \textit{then return } (\mathcal{S}, \textbf{error}) \\
& & \quad\quad (s, t') = \mathcal{E}valType(\mathcal{S}, t_2) \\
& & \quad\quad \textit{return } (s, t_1 \to t') \\
& & \quad \textit{elif } t_1 \equiv \textbf{error } \textit{then return } (\mathcal{S}, \textbf{error}) \\
& & \quad \textit{else} \\
& & \quad\quad (s, t') = \mathcal{E}valType(\mathcal{S}, t_1) \\
& & \quad\quad \textit{return } (s, t' \to t_2) \\
& & \textit{elif } t \equiv t_1 \textbf{ in } \{e_1, \cdots, e_n\} \textit{ then} \\
& & \quad \textit{if } \textbf{NormalType}(t_1) \textit{ then} \\
& & \quad\quad \textbf{for } i \in 1, \cdots, n \textbf{ do} \\
& & \quad\quad\quad \textit{if } \textbf{Normal}(e_i) == \textbf{false } \textit{then} \\
& & \quad\quad\quad\quad \textit{if } e_i \equiv \textbf{error } \textit{then return } (\mathcal{S}, \textbf{error}) \\
& & \quad\quad\quad\quad (s, e') = \mathcal{E}val(\mathcal{S}, e_i) \\
& & \quad\quad\quad\quad \textit{return } (s, t_1 \textbf{ in } \{\cdots, e_{i-1}, e', e_{i+1}, \cdots\}) \\
& & \quad\quad \textbf{done} \\
& & \quad\quad \textit{return } t \\
& & \quad \textit{elif } t_1 \equiv \textbf{error } \textit{then return } (\mathcal{S}, \textbf{error}) \\
& & \quad \textit{else} \\
& & \quad\quad (s, t') = \mathcal{E}valType(\mathcal{S}, t_1) \\
& & \quad\quad \textit{return } (s, t' \textbf{ in } \{e_1, \cdots, e_n\})
\end{array}
$$

$$\mathcal{E}valTypeAll \quad : \quad \mathcal{S} \times \boldsymbol{t} \to \mathcal{S} \times (\boldsymbol{t} \cup \{\textbf{error}\})$$

$$\begin{array}{l}\mathcal{E}valTypeAll(\mathcal{S},t) \;=\; \\ \qquad \textbf{while } (t \neq \textbf{error}) \textbf{ and } (\textbf{NormalType}(t) == \textbf{false}) \textbf{do} \\ \qquad\quad (\mathcal{S},t) = \mathcal{E}valType(\mathcal{S},t) \\ \qquad \textbf{done} \\ \qquad \textit{return } (\mathcal{S},t)\end{array}$$

4.5.4 Interprétation complète

Définition 4.53 *algorithme d'interprétation*

$$\mathcal{I}nterp \quad : \quad \mathcal{S} \times (\boldsymbol{c} \cup \boldsymbol{d}) \to \mathcal{S} \times \{\textbf{unit}, \textbf{error}\}$$

$$\begin{array}{l}\mathcal{I}nterp(\mathcal{S},p) \;=\; \textit{if } p \equiv c_1 \;\; c_2 \textit{ then} \\ \qquad (s,u) = \mathcal{I}nterp(\mathcal{S},c_1) \\ \qquad \textit{if } u \equiv \textbf{error} \textit{ then return } (s, \textbf{error}) \\ \qquad \textit{return } \mathcal{I}nterp(s,c_2) \\ \quad \textit{elif } p \equiv \textbf{module } x\{d\} \\ \qquad (s,u) = \mathcal{I}nterp(\mathcal{S},d) \\ \qquad \textit{if } u \equiv \textbf{error} \textit{ then return } (s, \textbf{error}) \\ \qquad \textit{if } s \equiv \mathcal{S} \cup s' \textit{ then} \\ \qquad\quad \textit{return } (\mathcal{S} \cup \{x = s'\}, u) \\ \quad \textit{elif } p \equiv d_1 \;\; d_2 \textit{ then} \\ \qquad (s,u) = \mathcal{I}nterp(\mathcal{S},d_1) \\ \qquad \textit{if } u \equiv \textbf{error} \textit{ then return } (s, \textbf{error}) \\ \qquad \textit{return } \mathcal{I}nterp(s,d_2) \\ \quad \textit{elif } p \equiv \textbf{from } x_1 \textbf{ import } x_2 \textbf{ as } x_3 \\ \qquad \textit{if } x_1 \notin \textit{ dom}(\mathcal{S}) \textit{ then return } \textbf{error} \\ \qquad \textit{elif } \mathcal{S}(x_1) = \mathcal{S}' \textit{ then} \\ \qquad\quad \textit{if } x_2 \notin \textit{ dom}(\mathcal{S}') \textit{ then return } \textbf{error} \\ \qquad\quad \textit{return } (\mathcal{S} \cup \{x_3 = \mathcal{S}'(x_2)\}, \textbf{unit}) \\ \qquad \textit{else return } \textbf{error} \\ \quad \textit{elif } (p \equiv \textbf{const } x : t = e) \textit{or } (p \equiv \textbf{const } x := e\} \textit{ then} \\ \qquad (s,v) = \mathcal{E}valAll(\mathcal{S},e) \\ \qquad \textit{if } v \equiv \textbf{error} \textit{ then return } (s, \textbf{error}) \\ \qquad \textit{return } (\mathcal{S} \cup \{x = v\}, \textbf{unit}) \\ \quad \textit{elif } p \equiv \textbf{type } \tau = t \textit{ then} \\ \qquad (s,u) = \mathcal{E}valTypeAll(\mathcal{S},t) \\ \qquad \textit{if } u \equiv \textbf{error} \textit{ then return } (s, \textbf{error}) \\ \qquad \textit{return } (\mathcal{S} \cup \{\tau = u\}, \textbf{unit})\end{array}$$

4.5.5 Propriétés

Il est important de démontrer que les algorithmes proposés reflètent bien le comportement exprimé par les règles de la sémantique opérationnelle, que se soit pour le calcul des termes ou des types. La

technique, classique, pour établir cette équivalence utilise deux étapes symétriques, la correction et la complétude. Notons que ces propositions tiennent compte des erreurs d'exécution.

Proposition 4.54 *Correction de l'évaluation des termes*

$$\begin{cases} \mathcal{E}valAll(\mathcal{S},e) = (\mathcal{S}',v) & \Rightarrow \quad [\mathcal{S}\vdash e] \twoheadrightarrow\!\circ\ [\mathcal{S}'\vdash v] \\ \mathcal{E}valAll(\mathcal{S},e) = (\mathcal{S}',\textbf{error}) & \Rightarrow \quad [\mathcal{S}\vdash e] \twoheadrightarrow [\mathcal{S}'\vdash \textbf{error}] \end{cases}$$

Preuve : Une première étape est de prouver un lemme sur la correction de la fonction $\mathcal{E}val$ par rapport aux transitions atomiques $[\mathcal{S}\vdash\,] \twoheadrightarrow [\mathcal{S}\vdash\,]$, par induction. L'étape suivante est directe, d'après la définition itérative de $\mathcal{E}valAll$ et la structure des règles [e-norm-t],[e-dir] et [e-trans] (cf 4.2.1, en introduction de la section sur la sémantique opérationnelle). □

Proposition 4.55 *Complétude de l'évaluation des termes*

$$\begin{cases} [\mathcal{S}\vdash e] \twoheadrightarrow\!\circ\ [\mathcal{S}'\vdash v] & \Rightarrow \quad \mathcal{E}valAll(\mathcal{S},e) = (\mathcal{S}',v) \\ [\mathcal{S}\vdash e] \twoheadrightarrow [\mathcal{S}'\vdash \textbf{error}] & \Rightarrow \quad \mathcal{E}valAll(\mathcal{S},e) = (\mathcal{S}',\textbf{error}) \end{cases}$$

Preuve : De manière équivalente, en deux étapes. □

Proposition 4.56 *Terminaison dans tous les cas de l'évaluation des termes*

$$\forall \mathcal{S}, e \quad \mathcal{E}valAll(\mathcal{S},e) = (\mathcal{S}',v) \textit{ ou } \mathcal{E}valAll(\mathcal{S},e) = (\mathcal{S}',\textbf{error})$$

Preuve : Dans la boucle principale, l'invariant est évident à établir. Dans la fonction $\mathcal{E}val$, on remarque que tout appel récursif s'accompagne d'une réduction de la complexité structurelle de l'argument. La fonction qui mesure ce critère (par exemple une simple pondération additive des sous termes) est monotone décroissante, ce qui assure la terminaison dans tous les cas. □

Proposition 4.57 *Correction de l'évaluation des types*

$$\begin{cases} \mathcal{E}valTypeAll(\mathcal{S},t) = (\mathcal{S}',u) & \Rightarrow \quad [\mathcal{S}\vdash t] \twoheadrightarrow\!\circ\ [\mathcal{S}'\vdash u] \\ \mathcal{E}valTypeAll(\mathcal{S},t) = (\mathcal{S}',\textbf{error}) & \Rightarrow \quad [\mathcal{S}\vdash t] \twoheadrightarrow [\mathcal{S}'\vdash \textbf{error}] \end{cases}$$

Preuve : De manière équivalente, en deux étapes. □

Proposition 4.58 *Complétude de l'évaluation des types*

$$\begin{cases} [\mathcal{S}\vdash t] \twoheadrightarrow\!\circ\ [\mathcal{S}'\vdash u] & \Rightarrow \quad \mathcal{E}valTypeAll(\mathcal{S},t) = (\mathcal{S}',u) \\ [\mathcal{S}\vdash t] \twoheadrightarrow [\mathcal{S}'\vdash \textbf{error}] & \Rightarrow \quad \mathcal{E}valTypeAll(\mathcal{S},t) = (\mathcal{S}',\textbf{error}) \end{cases}$$

Preuve : De manière équivalente, en deux étapes. □

Proposition 4.59 *Terminaison dans tous les cas de l'évaluation des types*

$$\forall \mathcal{S}, t \quad \mathcal{E}valTypeAll(\mathcal{S},t) = (\mathcal{S}',u) \textit{ ou } \mathcal{E}valTypeAll(\mathcal{S},t) = (\mathcal{S}',\textbf{error})$$

Preuve : Même remarque que pour la terminaison de l'évaluation des termes. D'autre part, cette dernière propriété permet de plus d'assurer la terminaison dans tous les cas de l'évaluation des types (concerne le calcul des types énumérés). □

4.6 Contrôle des types

Dans cette section, nous proposons un algorithme qui permet de réaliser concrètement le contrôle des types. Nous demontrons ensuite que l'algorithme est correct (les types controlés comme valides sont corrects par rapport au système de types), complet (toute expression correctement typée peut être contrôlée avec succès), et qu'il termine dans tous les cas. L'algorithme est présenté sous la forme d'une fonction principale

$$\mathcal{C}hk : \boldsymbol{\gamma} \times \boldsymbol{e} \times \boldsymbol{t} \rightarrow \{\textbf{true}, \textbf{false}\}$$

de trois autres fonctions,

$$\mathcal{S}ub : \boldsymbol{\gamma} \times \boldsymbol{t} \times \boldsymbol{t} \rightarrow \{\textbf{true}, \textbf{false}\}$$

et

$$\mathcal{T}ype : \boldsymbol{\gamma} \times \boldsymbol{t} \rightarrow \{\textbf{true}, \textbf{false}\}$$

et enfin

$$\mathcal{F}nd : \boldsymbol{\gamma} \times \boldsymbol{e} \rightarrow \boldsymbol{t} \cup \{\textbf{error}\}$$

respectivement pour décider la relation de sous-typage, décider si un type est bien formé, et enfin inférer un type minimal pour toute expression. Nous devons encore leur adjoindre une fonction

$$\mathcal{A}ddType : \boldsymbol{\gamma} \times \boldsymbol{t} \times \boldsymbol{t} \rightarrow \boldsymbol{t}$$

qui calcule l'union minimale de deux types, et une fonction

$$\mathcal{P}rodType : \boldsymbol{\gamma} \times \boldsymbol{t} \times \boldsymbol{t} \rightarrow \boldsymbol{t}$$

qui calcule l'intersection maximale. Notons que les six algorithmes cités sont fonctionnellement liés par des appels récursifs croisés. La fonction

$$\mathit{conv}(\ ) : \boldsymbol{\gamma} \times \boldsymbol{e} \rightarrow \{\textbf{true}, \textbf{false}\}$$

réalise le test syntaxique de confluence sur tout terme e bien typé, et enfin

$$\mathcal{M}kEnv : \boldsymbol{\gamma} \rightarrow \mathbb{S}$$

Calcule un environnement d'exécution conforme à tout context de typage bien formé. Les notations $\boldsymbol{\gamma}, \boldsymbol{e}$ et $\boldsymbol{t}$ dénotent respectivement les ensembles de tous les environnements, toutes les expressions et tous les types.

4.6.1 Validation de confluence

Le premier algorithme permet de restreindre les termes pouvant être énumérés dans un type, au moyen d'un contrôle syntaxique qui permet de détecter les termes susceptibles ne pas être confluents (par défaut de déterminisme). Un test syntaxique est assez "grossier" en ce sens qu'il peut éliminer des termes confluents, mais il est suffisant et réaliste en pratique. Notons le traitement des lambda applications : si le paramètre est confluent, alors la confluence du corps de la lambda fonction est testée après substitution du paramètre. Toutefois, il ne s'agit pas d'une β-réduction, car le paramètre n'est pas normalisé avant substitution. Notons encore que si ce test parait inutile au stade courant de définition du langage (tous les termes du noyau sont confluents), il constitue un squelette sur lequel d'autres tests plus pertinents viendront se rattacher lors des extensions décrites plus avant.

Définition 4.60 *Algorithme de test syntaxique de confluence*

conv(,) : $\gamma \times e \rightarrow \{\textbf{true}, \textbf{false}\}$

conv(γ, e) =

if **Normal**(e) *then return* **true**
elif $(e \equiv \textbf{x})$ *then*
  if $x : t = v \in \gamma$ *then return* **true**
  else return **false**
elif $(e \equiv e_1$ **and** $e_2)$ *then return* conv(γ, e_1) *and* conv(γ, e_2)
elif $(e \equiv$ **not** $e_1)$ *then return* conv(γ, e_1)
elif $(e \equiv e_1 + e_2)$ *then return* conv(γ, e_1) *and* conv(γ, e_2)
elif $(e \equiv e_1 \leq e_2)$ *then return* conv(γ, e_1) *and* conv(γ, e_2)
elif $(e \equiv e_1 == e_2)$ *then return* conv(γ, e_1) *and* conv(γ, e_2)
elif $(e \equiv$ **if** e_1 **then** e_2 **else** $e_3)$ *then return* conv(γ, e_1) *and* conv(γ, e_2) *and* conv(γ, e_3)
elif $(e \equiv \lambda x : t.e_1(e_2))$ *then return* conv(γ, e_2) *and* conv$(\gamma, e_1[e_2/x])$
elif $(e \equiv \textbf{x}(e_2))$ *then return* conv$(\gamma, \gamma(x)(e_2))$

Le lemme suivant établit que tout terme bien typé et déclaré confluent par l'algorithme de test converge nécessairement vers une valeur normale unique.

Lemme 4.61 *Correction du test syntaxique de convergence*

$$\gamma \rhd e : \top \textit{ et } \text{conv}(\gamma, e) == \textbf{true} \quad \Rightarrow \quad \textit{il existe } v \textit{ tel que } \quad \gamma \rhd e \rightsquigarrow v$$

Preuve : Par induction sur la structure des termes e, et d'après la définition de la confluence [te-conv]. Le cas le plus délicat se pose pour la lambda-application. Il faut alors penser à utiliser l'ordre présent dans l'avant dernier test de l'algorithme : celui-ci implique que le test de confluence est propagé au corps de la fonction uniquement si le paramètre est lui-même confluent. L'hypothèse que le terme est bien typé est notamment utilisée dans le cas du dernier test algorithmique (lambda-application effectuée au moyen d'une variable). □

4.6.2 Création d'un environnement d'exécution conforme

L'algorithme de contrôle de type fait nécessairement appel à un évaluateur (types énumérés). De plus l'évaluation doit se dérouler dans un environnement d'exécution conforme au contexte de typage. L'algorithme suivant se propose de construire un environnement d'exécution conforme à un contexte de typage γ quelconque (toutefois, on suppose implicitement que ce dernier est bien formé). Notons que les entrées non valuées dans le contexte de typage (notées par exemple $\gamma, x : t$) ne sont pas représentées dans le contexte d'exécution.

Définition 4.62 *Algorithme de construction d'un environnement conforme*

$$\mathcal{M}kEnv \quad : \quad \gamma \rightarrow \mathcal{S}$$

$\mathcal{M}kEnv(\gamma)$ =

$\mathcal{S} = \{\ \}$
for $x \in$ *dom*(γ) **do**
  if $(\gamma(x) \equiv x : t = v)$ *then* $\mathcal{S} = \mathcal{S} \cup \{x = v\}$
  elif $(\gamma(x) \equiv x :: t)$ *then* $\mathcal{S} = \mathcal{S} \cup \{x = t\}$
  elif $(\gamma(x) \equiv x :: \gamma')$ *then* $\mathcal{S} = \mathcal{S} \cup \{x = \mathcal{M}kEnv(\gamma')\}$
done
return $\mathcal{S}$

Le lemme suivant affirme que l'environnement calculé par l'algorithme précédent est en conformité avec le contexte de typage.

Lemme 4.63 *Correction d'un environnement d'exécution dérivé d'un contexte de typage*

$$\mathcal{M}kEnv(\gamma) == \mathcal{S} \quad \Rightarrow \quad \Vdash \mathcal{S} : \gamma$$

Preuve : L'emboîtement récursif des contextes impose l'utilisation d'une technique d'induction (bien que de part le langage même, seul un unique niveau de récursion est pertinent ; cf les règles de syntaxe et sémantique concernant les déclarations de modules). Démonstration sans difficultés particulières. □

4.6.3 Types bien formés

L'algorithme suit assez directement la définition logique. Toutefois, on notera l'utilisation en "interne" d'un algorithme de vérification de termes bien typés, défini plus loin, ainsi que du test de confluence déjà présenté. La notation $\gamma(\tau)$ signifie que l'on résoud (déréférence) la variable de type τ au moyen du contexte de typage. Notons que ce dernier est supposé bien formé, c'est a dire toujours capable de résoudre la référence.

Définition 4.64

$$\mathcal{T}ype \quad : \quad \gamma \times t \rightarrow \{\textbf{true}, \textbf{false}\}$$

$\mathcal{T}ype(\gamma, t)$ =

if $t \in \{\bot, \top, \textbf{None}, \textbf{num}, \textbf{string}, \textbf{bool}\}$ *then return* **true**
if $t \equiv \tau$ *then return* $\mathcal{T}ype(\gamma, \gamma(t))$
if $(t \equiv t_1$ **in** $\{e_1, \cdots, e_n\})$ *then*
  if $(n > 0)$ *and* $\mathcal{T}ype(\gamma, t_1)$ *then*
    for $e \in e_1 \cdots e_n$ **do**
      if conv$(\gamma, e) =$ **false** *or* $\mathcal{C}hk(\gamma, e, t_1) ==$ **false** *then*
        return **false**
    done
    return **true**
  else return **false**
if $(t \equiv t_1 \rightarrow t_2)$ *then return* $\mathcal{T}ype(\gamma, t_1)$ *and* $\mathcal{T}ype(\gamma, t_2)$
return **false**

4.6.4 Sous-typage

Définition 4.65 *Algorithme de décision sur la relation de sous-type*

$$\mathcal{S}ub \quad : \quad \boldsymbol{\gamma} \times \boldsymbol{t} \times \boldsymbol{t} \rightarrow \{\textbf{true}, \textbf{false}\}$$

$\mathcal{S}ub(\gamma, t_1, t_2)$ =

if $(t_1 \equiv \tau)$ *then return* $\mathcal{S}ub(\gamma, \gamma(\tau), t_2)$
if $(t_2 \equiv \tau)$ *then return* $\mathcal{S}ub(\gamma, t_1, \gamma(\tau))$
if $(t_1 \equiv \bot)$ *then return* **true**
if $(t_1 \equiv$ **None**$)$ *then*
  if $\mathcal{T}ype(\gamma, t_2)$ *then*
    if $t_2 \equiv t$ **in** $\{e_1, \cdots, e_n\}$ *then return* $\mathcal{C}hk(\gamma,$ **none**$, t_2)$
    if $t_2 \notin \{\bot, t_{21} \rightarrow t_{22}\}$*return* **true**
    return **false**
  return **false**
if $(t_1 \equiv \top)$ *then return* $(t_2 \equiv \top)$
if $(t_1 \in \{$**num**, **string**, **bool**, **Unit**$\})$ *then return* $(t_2 \equiv t_1)$ *or* $(t_2 \equiv \top)$
if $(t_1 \equiv t_{11} \rightarrow t_{12})$ *then*
  if $(t_2 \equiv t_{21} \rightarrow t_{22})$ *then*
    if $\mathcal{T}ype(t_1)$ *and* $\mathcal{T}ype(t_2)$ *then return* $\mathcal{S}ub(\gamma, t_{12}, t_{22})$ *and* $\mathcal{S}ub(\gamma, t_{21}, t_{11})$
    return **false**
  if $(t_2 \equiv \top)$ *then return* **true**
  return **false**
if $t_1 \equiv t$ **in** $\{e_1, \cdots, e_n\}$ *then*
  if $\mathcal{T}ype(t_1) ==$ **false or** $\mathcal{T}ype(t_2) ==$ **false** *then return* **false**
  for e **in** $e_1 \cdots e_n$ **do**
    if $\mathcal{C}hk(\gamma, e, t_2) ==$ **false** *then return* **false**
  done ;
  return **true**
return **false**

L'algorithme testant l'égalité structurelle de deux types est la traduction directe de la définition.

Définition 4.66 *Algorithme de décision pour l'équivalence de types*

$$\mathcal{E}qType \quad : \quad \boldsymbol{\gamma} \times \boldsymbol{t} \times \boldsymbol{t} \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\mathcal{E}qType(\gamma, t_1, t_2) \quad = \quad \mathcal{S}ub(\gamma, t_1, t_2) \textit{ and } \mathcal{S}ub(\gamma, t_2, t_1)$$

4.6.5 Union et intersection de types

Ces deux algorithmes utilisent la commutativité des opérateurs $\oplus$ et $\otimes$ pour simplifier le traitement des types énumérés. La contravariance liée aux types fonction se traduit par une récursivité croisée pour chaque algorithme.

Définition 4.67 *Algorithme de calcul pour l'opérateur* $\oplus$

$$\begin{array}{lll} \mathcal{A}ddType & : & \boldsymbol{\gamma \times t \times t \to t} \\ \\ \mathcal{A}ddType(\gamma, t_1, t_2) & = & \\ & 1 & \textit{if } t_1 \equiv \tau \textit{ then return } \mathcal{A}ddType(\gamma, \gamma(\tau), t_2) \\ & 2 & \textit{if } t_2 \equiv \tau \textit{ then return } \mathcal{A}ddType(\gamma, t_1, \gamma(\tau)) \\ & 3 & \textit{if } \mathcal{S}ub(\gamma, t_1, t_2) \textit{ then return } t_2 \\ & 4 & \textit{if } \mathcal{S}ub(\gamma, t_2, t_1) \textit{ then return } t_1 \\ & 5 & \textit{if } t_1 \equiv t_{11} \textbf{ in } S_1 \textit{ then} \\ & 6 & \quad \textit{if } t_2 \equiv t_{12} \textbf{ in } S_2 \textit{ then return } \top \textbf{ in } S_1 \cup S_2 \\ & 7 & \quad \textit{if } (t_2 \equiv \textbf{None}) \textit{ then} \\ & 8 & \quad\quad \textit{if } \mathcal{S}ub(\gamma, \textbf{None}, t_1) \textit{ then } Return \top \textbf{ in } S_1, \{\textbf{none}\} \\ & 9 & \quad \textit{return } \top \\ & 10 & \textit{if } t_2 \equiv t_{21} \textbf{ in } S_2 \textit{ then return } \mathcal{A}ddType(\gamma, t_2, t_1) \\ & 11 & \textit{if } t_1 \equiv t_{11} \to t_{12} \textit{ and } t_2 \equiv t_{21} \to t_{22} \textit{ then} \\ & 12 & \quad \textit{return } \mathcal{P}rodType(\gamma, t_{11}, t_{21}) \to \mathcal{A}ddType(\gamma, t_{12}, t_{22}) \\ & 13 & \textit{return } \top \end{array}$$

Cet algorithme est légèrement plus complexe, car dans le cas où le premier type argument t_1 est énuméré, chaque terme constitutif est testé (cf lignes 5-11). Si aucun d'eux ne possède le type t_2, alors le résultat est le type nul $\perp$. Dans le cas contraire, le résultat est un type énuméré constitué de tous les termes ayant le type t_2.

Définition 4.68 *Algorithme de calcul pour l'opérateur* $\otimes$

$$\begin{array}{lll} \mathcal{P}rodType & : & \boldsymbol{\gamma \times t \times t \to t} \\ \\ \mathcal{P}rodType(\gamma, t_1, t_2) & = & \\ & 1 & \textit{if } t_1 \equiv \tau \textit{ then return } \mathcal{P}rodType(\gamma, \gamma(\tau), t_2) \\ & 2 & \textit{if } t_2 \equiv \tau \textit{ then return } \mathcal{P}rodType(\gamma, t_1, \gamma(\tau)) \\ & 3 & \textit{if } \mathcal{S}ub(\gamma, t_1, t_2) \textit{ then return } t_1 \\ & 4 & \textit{if } \mathcal{S}ub(\gamma, t_2, t_1) \textit{ then return } t_2 \\ & 5 & \textit{if } t_1 \equiv t_{11} \textbf{ in } S_1 \textit{ then} \\ & 6 & \quad S = \emptyset \\ & 7 & \quad \textbf{for } e \in S_1 \textbf{ do} \\ & 8 & \quad\quad \textit{if } \mathcal{C}hk(\gamma, e, t_2) \textit{ then } S = S \cup \{e\} \\ & 9 & \quad \textbf{done} \\ & 10 & \quad \textit{if } S \neq \emptyset \textit{ then return } \top \textbf{ in } s \\ & 11 & \quad \textit{return } \perp \\ & 12 & \textit{if } t_2 \equiv t_{21} \textbf{ in } S_2 \textit{ then return } \mathcal{P}rodType(\gamma, t_2, t_1) \\ & 13 & \textit{if } t_1 \equiv t_{11} \to t_{12} \textit{ and } t_2 \equiv t_{21} \to t_{22} \textit{ then} \\ & 14 & \quad \textit{return } \mathcal{A}ddType(\gamma, t_{11}, t_{21}) \to \mathcal{P}rodType(\gamma, t_{12}, t_{22}) \\ & 15 & \textit{return } \perp \end{array}$$

4.6.6 Inférence de type

Cet algorithme est certainement le plus complexe de tous. A sa base, tout litteral de type primitif **num**, **string**, **bool** se voit inférer comme type minimal l'énumération constituée de lui-même comme

unique élément. D'autre part, l'algorithme est en charge de propager les types minimaux dans les termes complexes, en respectant les équations de typage minimal étudiées dans le cadre du système formel.

Définition 4.69 *Algorithme de calcul du type minimal d'un terme*

$\mathcal{F}nd \quad : \quad \boldsymbol{\gamma} \times \boldsymbol{e} \rightarrow \boldsymbol{t} \cup \{\textbf{error}\}$

$\mathcal{F}nd(\gamma, e) \quad =$

if $(e \equiv$ **none**$)$ *then return* **None in** $\{$**none**$\}$
if $(e \equiv$ **unit**$)$ *then return* **Unit in** $\{$**unit**$\}$
if $(e \equiv$ **n**$)$ *then return* **num in** $\{e\}$
if $(e \equiv$ **s**$)$ *then return* **string in** $\{e\}$
if $(e \equiv$ **x**$)$ *then return* $\gamma(x)$
if $(e \in \{$**true**, **false**$\})$ *then return* **bool in** $\{e\}$
elif $(e \equiv e_1$ **and** $e_2)$ *then*
  $t_1 = \mathcal{F}nd(\gamma, e_1)$
  if $t_1 =$ **error** *or* $\mathcal{S}ub(\gamma, t_1,$ **bool**$) =$ **false** *then return* **error**
  $t_2 = \mathcal{F}nd(\gamma, e_2)$
  if $t_2 =$ **error** *or* $\mathcal{S}ub(\gamma, t_2,$ **bool**$) =$ **false** *then return* **error**
  if $t_1 \equiv t_{11}$ **in** $\{e_1, \cdots, e_n\}$ *then*
    if $t_2 \equiv t_{21}$ **in** $\{e'_1, \cdots, e'_k\}$ *then*
      $t = \perp$
      $\mathcal{S} = \mathcal{M}kEnv(\gamma)$
      for $e \in e_1 \cdots e_n$ **do**
        for $e' \in e'_1 \cdots e'_k$ **do**
          $(s, v) = \mathcal{E}valAll(\mathcal{S}, e$ **and** $e')$
          $t = \mathcal{A}ddType(\gamma, t,$ **bool in** $\{v\})$
        done
      done
      return t
    if $t_2 \equiv$ **bool** *then*
      if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**true**$\})$ *then return* **bool**
      if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**none**$\})$ *then return* t_1
      if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**true**, **false**$\})$ *then return* **bool**
      if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**true**, **none**$\})$ *then return* **bool**
  if $t_1 \equiv$ **bool** *then*
    if $\mathcal{S}ub(\gamma, t_2,$ **bool in** $\{$**true**$\})$ *then return* **bool**
    if $\mathcal{S}ub(\gamma, t_2,$ **bool in** $\{$**none**$\})$ *then return* **bool in** $\{$**false**, **none**$\}$
    if $\mathcal{S}ub(\gamma, t_2,$ **bool in** $\{$**false**$\})$ *then return* **bool in** $\{$**false**, **none**$\}$
    if $\mathcal{S}ub(\gamma, t_2,$ **bool in** $\{$**true**, **false**$\})$ *then return* **bool**
    if $\mathcal{S}ub(\gamma, t_2,$ **bool in** $\{$**false**, **none**$\})$ *then return* t_2
    return **bool**
if $(e \equiv e_1 \leq e_2)$ *then*
  if $\mathcal{C}hk(\gamma, e_1,$ **num**$)$ *and* $\mathcal{C}hk(\gamma, e_2,$ **num**$)$ *then return* **bool**
  if $\mathcal{C}hk(\gamma, e_1,$ **string**$)$ *and* $\mathcal{C}hk(\gamma, e_2,$ **string**$)$ *then return* **bool**
  return **error**

```
if (e ≡ not e') then
   t1 = Fnd(γ, e')
   if t1 = error or Sub(γ, t1, bool) = false then return error
   if t1 ≡ t11 in {e1, ··· , en} then
      t = ⊥; S = MkEnv(γ)
      for e'' ∈ e1 ··· en do
         (s, v) = EvalAll(S, not e'')
         t = AddType(γ, t, bool in {v})
      done
      return t
   return bool
if (e ≡ e1 + e2) then
   t1 = Fnd(γ, e1)
   if t1 = error then return error
   if Sub(γ, t1, num) then
      t2 = Fnd(γ, e2)
      if t2 = error or Sub(γ, t2, num) = false then return error
      if t1 ≡ t11 in {e1, ··· , en}) then
         if t2 ≡ t12 in {e'1, ··· , e'k}) then
            t = ⊥; S = MkEnv(γ)
            for e ∈ e1 ··· en do
               for e' ∈ e'1 ··· e'k do
                  v = EvalAll(S, e + e')
                  t = AddType(γ, t, num in {v})
               done
            done
            return t
         return num
      return num
   if Sub(γ, t1, string)
      t2 = Fnd(γ, e2)
      if t2 = error or Sub(γ, t2, string) = false then return error
      if t1 ≡ t11 in {e1, ··· , en}) then
         if t2 ≡ t12 in {e'1, ··· , e'k}) then
            t = ⊥; S = MkEnv(γ)
            for e ∈ e1 ··· en do
               for e' ∈ e'1 ··· e'k do
                  (s, v) = EvalAll(S, e + e')
                  t = AddType(γ, t, string in {v})
               done
            done
            return t
         return string
      return string
   return error
```

elif $(e \equiv e_1 == e_2)$ *then*
  $t_1 = \mathcal{F}nd(\gamma, e_1)$
  if $t_1 =$ **error** *then return* **error**
  $t_2 = \mathcal{F}nd(\gamma, e_2)$
  if $t_2 =$ **error** *then return* **error**
  if $t_1 \equiv t_{11}$ **in** $\{e_1, \cdots, e_n\}$ *then*
    if $t_2 \equiv t_{21}$ **in** $\{e'_1, \cdots, e'_k\}$ *then*
      $diffVal = 0$
      $eqVal = 0$
      for $e \in e_1 \cdots e_n$ **do**
        for $e' \in e'_1 \cdots e'_k$ **do**
          $(s, v) = \mathcal{E}valAll(\mathcal{S}, e == e')$
          if $v \equiv$ **true** *then* $eqval = eqval + 1$
          else $diffval = diffval + 1$
        done
      done
      if $eqval > 0$ *and* $diffVal == 0$ *then return* **bool in** $\{$**true**$\}$
      if $eqval == 0$ *then return* **bool in** $\{$**false**$\}$
      return **bool in** $\{$**false**, **true**$\}$
    return **bool in** $\{$**false**, **true**$\}$
  if $t_1 \equiv$ **None** *and* $t_2 \equiv$ **None** *then return* **bool in** $\{$**true**$\}$
  return **bool in** $\{$**false**, **true**$\}$
if $(e \equiv$ **if** e_1 **then** e_2 **else** $e_3)$ *then*
  $t_1 = \mathcal{F}nd(\gamma, e_1)$
  if $t_1 =$ **error** *or* $\mathcal{S}ub(\gamma, t_1,$ **bool**$) ==$ **false** *then return* **error**
  $t_2 = \mathcal{F}nd(\gamma, e_2)$
  if $t_2 =$ **error** *then return* **error**
  $t_3 = \mathcal{F}nd(\gamma, e_3)$
  if $t_3 =$ **error** *then return* **error**
  if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**true**$\})$ *then return* t_2
  if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**false**$\})$ *then return* t_3
  if $\mathcal{S}ub(\gamma, t_1,$ **None**$)$ *then return* **None**
  if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**true**, **false**$\})$ *then return* $\mathcal{A}ddType(\gamma, t_2, t_3)$
  if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**true**, **none**$\})$ *then return* $\mathcal{A}ddType(\gamma, t_2,$ **None**$)$
  if $\mathcal{S}ub(\gamma, t_1,$ **bool in** $\{$**false**, **none**$\})$ *then return* $\mathcal{A}ddType(\gamma, t_3,$ **None**$)$
  return $\mathcal{A}ddType(\gamma, t_2, \mathcal{A}ddType(\gamma, t_3,$ **None**$))$
elif $(e \equiv \lambda x : t.e_1)$ *then*
  if $x \in$ *dom*(γ) *then return* **error**
  if $\mathcal{T}ype(\gamma, t) =$ **false** *then return* **error**
  $r = \mathcal{F}nd(\gamma \cup \{x : t\}, e_1)$
  if $r =$ **error** *then return* **error**
  else return $(t \rightarrow r)$
elif $e \equiv e_1(e_2)$ *then*
  if $\mathcal{F}nd(\gamma, e_1) \equiv (t_1 \rightarrow t_2)$ *then*
    if $\mathcal{C}hk(\gamma, e_2, t_1)$ *then return* t_2
    else return **error**
  else return **error**

4.6.7 Conformité de types

Pour un terme et un type donné, dans un contexte γ bien formé, l'algorithme détermine si le terme à bien le type spécifié. Notons le traitement des types énumérés par utilisation du test de confluence, puis l'évaluation dans le contexte d'exécution construit à partir de γ, afin de calculer la forme normale..

Definition

$$\mathcal{C}hk \quad : \quad \boldsymbol{\gamma} \times \boldsymbol{e} \times \boldsymbol{t} \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\begin{array}{l}
\mathcal{C}hk(\gamma, e, t) \quad = \\
\quad \textit{if } \mathcal{F}nd(\gamma, e) \neq \textbf{error } \textit{then} \\
\quad\quad \textit{if } t \equiv \tau \textit{ then return } \mathcal{C}hk(\gamma, e, \gamma(\tau)) \\
\quad\quad \textit{elif } t \equiv t_1 \textbf{ in } \{e_1, \cdots, e_n\} \textit{ then} \\
\quad\quad\quad \textit{if conv}(\gamma, e) \textit{ then} \\
\quad\quad\quad\quad \mathcal{S} = \mathcal{M}kEnv(\gamma) \\
\quad\quad\quad\quad s, v = \mathcal{E}valAll(\mathcal{S}, e) \\
\quad\quad\quad\quad \textbf{for } e' \textbf{ in } e_1 \cdots e_n \textbf{ do} \\
\quad\quad\quad\quad\quad \textit{if } \mathcal{C}hk(\gamma, e', t_1) \textit{ and conv}(\gamma, e') \textit{ then} \\
\quad\quad\quad\quad\quad\quad \textit{if } \mathcal{E}valAll(\mathcal{S}, e') == (s, v) \\
\quad\quad\quad\quad\quad\quad\quad \textit{then return } \textbf{true} \\
\quad\quad\quad\quad\quad \textit{else return } \textbf{false} \\
\quad\quad\quad\quad \textbf{done} \\
\quad\quad\quad\quad \textit{return } \textbf{false} \\
\quad\quad\quad \textit{else return } \textbf{false} \\
\quad\quad \textit{elif } e \equiv \textbf{n } \textit{then} \\
\quad\quad\quad \textit{if } t \in \{\textbf{num}, \top\} \textit{ then return } \textbf{true} \\
\quad\quad\quad \textit{else return } \textbf{false} \\
\quad\quad \textit{elif } e \equiv \textbf{s } \textit{then} \\
\quad\quad\quad \textit{if } t \in \{\textbf{string}, \top\} \textit{ then return } \textbf{true} \\
\quad\quad\quad \textit{else return } \textbf{false} \\
\quad\quad \textit{elif } e \in \{\textbf{true}, \textbf{false}\} \textit{ then} \\
\quad\quad\quad \textit{if } t \in \{\textbf{bool}, \top\} \textit{ then return } \textbf{true} \\
\quad\quad\quad \textit{else return } \textbf{false} \\
\quad\quad \textit{elif } e \equiv \textbf{unit } \textit{then} \\
\quad\quad\quad \textit{if } t \in \{\textbf{Unit}, \top\} \textit{ then return } \textbf{true} \\
\quad\quad\quad \textit{else return } \textbf{false} \\
\quad\quad \textit{elif } e \equiv \textbf{none } \textit{then return } \mathcal{T}ype(\gamma, t) \\
\quad\quad \textit{else return } \mathcal{S}ub(\gamma, \mathcal{F}nd(\gamma, e), t) \\
\quad \textit{else return } \textbf{false}
\end{array}$$

4.6.8 Propriétés

Ici encore, nous souhaitons prouver que les algorithmes sont bien en étroite relation avec le système de type formel : Correction et complétude sont une fois encore à établir. La difficulté vient du schéma

de récursion très complexe, qui ne compte pas moins de quatre ou cinq appels récursifs croisés. De plus, l'algorithme $\mathcal{C}hk$ utilise à la fois un terme et un type, d'où une induction plus complexe. La technique, developpée en annexe, utilise une double induction sur la structure des termes et des types, menée en parallèle sur l'ensemble des algorithmes (ils sont en effet tous fonctionnellement reliés par les appels récursifs). D'autre part, les lemmes et diverses propriétés caractérisant les algorithmes d'évaluation sont mis à contribution : test de confluence, construction de contexte d'exécution conforme, correction et complétude de l'évaluation des termes.

Proposition 4.70 *Correction*

$$\begin{cases} \mathcal{T}ype(\gamma,t) \equiv \textbf{true} & \Rightarrow \quad \gamma \ \rhd \ t \\ \mathcal{S}ub(\gamma,t_1,t_2) \equiv \textbf{true} & \Rightarrow \quad \gamma \rhd t_1 \preccurlyeq t_2 \\ \mathcal{A}ddType(\gamma,t_1,t_2) \equiv t & \Rightarrow \quad \gamma \rhd t \doteq t_1 \oplus t_2 \\ \mathcal{P}rodType(\gamma,t_1,t_2) \equiv t & \Rightarrow \quad \gamma \rhd t \doteq t_1 \otimes t_2 \\ \gamma \rhd t \doteq \mathcal{F}nd(\gamma,e) & \Rightarrow \quad \gamma \rhd e \preccurlyeq: t \\ \mathcal{C}hk(\gamma,e,t) \equiv \textbf{true} & \Rightarrow \quad \gamma \rhd e : t \end{cases}$$

Proposition 4.71 *Complétude*

$$\begin{cases} \gamma \ \rhd \ t & \Rightarrow \quad \mathcal{T}ype(\gamma,t) \equiv \textbf{true} \\ \gamma \rhd t_1 \preccurlyeq t_2 & \Rightarrow \quad \mathcal{S}ub(\gamma,t_1,t_2) \equiv \textbf{true} \\ \gamma \rhd t \doteq t_1 \oplus t_2 & \Rightarrow \quad \gamma \rhd t \doteq \mathcal{A}ddType(\gamma,t_1,t_2) \\ \gamma \rhd t \doteq t_1 \otimes t_2 & \Rightarrow \quad \gamma \rhd t \doteq \mathcal{P}rodType(\gamma,t_1,t_2) \\ \gamma \rhd e : t & \Rightarrow \quad \mathcal{C}hk(\gamma,e,t) \equiv \textbf{true} \\ \gamma \rhd e \preccurlyeq: t & \Rightarrow \quad \gamma \rhd t \doteq \mathcal{F}nd(\gamma,e) \end{cases}$$

Preuve : Developpée en annexe. □

Il nous reste encore à prouver que l'algorithme termine dans tous les cas. En effet, la complétude implique la terminaison lorsque les relations logiques sont vérifiées, comme par exemple, s'il est vrai que le terme e a pour type t. Cependant, nous ne savons encore rien sur le comportement de l'algorithme lorsque les hypothèses logiques sont fausses.

Proposition 4.72 *Terminaison dans tous les cas*

Pour tout environnement bien formé $\gamma, \forall t, e, t_1, t_2$

$$\begin{cases} \mathcal{T}ype(\gamma,t) \equiv \textbf{true} \textit{ ou } \mathcal{T}ype(\gamma,t) \equiv \textbf{false} \\ \mathcal{S}ub(\gamma,t_1,t_2) \equiv \textbf{true} \textit{ ou } \mathcal{S}ub(\gamma,t_1,t_2) \equiv \textbf{false} \\ \exists t_3 \textit{ tel que} \mathcal{A}ddType(\gamma,t_1,t_2) \equiv t_3 \\ \exists t_3 \textit{ tel que} \mathcal{P}rodType(\gamma,t_1,t_2) \equiv t_3 \\ \mathcal{C}hk(\gamma,e,t) \equiv \textbf{true} \textit{ ou } \mathcal{C}hk(\gamma,e,t) \equiv \textbf{false} \\ (\exists t_3 \textit{ tel que} \mathcal{F}nd(\gamma,e) \equiv t_3) \textit{ ou } \mathcal{F}nd(\gamma,e) \equiv \textbf{error} \end{cases}$$

Preuve : Utilise trois remarques : les itérations mises en œuvre dans l'algorithme sont toutes finies par construction. D'autre part, tout appel récursif, croisé ou non, abaisse la complexité du terme ou type passé en paramètre (terme ou type considéré comme fermé, et dont la structure est toujours finie par construction). Ainsi toute fonction qui mesure la complexité structurelle des termes lors des récursions est monotone décroissante. Ce qui implique nécessairement que les récursions prennent fin avec les feuilles terminales des termes ou types. La troisième remarque concerne les évaluations intermédiaires (cf $\mathcal{C}hk$, cas d'un type énuméré, par exemple), qui terminent toujours par propriété du test de confluence réalisé systématiquement avant évaluation (cf le lemme 4.61). □

4.7 Synthèse

Ce chapitre nous a permis de définir formellement le cœur du langage *Circus*, mais aussi de le caractériser mathématiquement. Au-delà de la syntaxe et de la sémantique opérationnelle, un système de type original, extension du lambda-calcul typé intégrant un type énuméré et une relation de sous-typage a été défini au moyen d'un système d'inférence logique. Dans un second temps, le système de type a été prouvé correct par rapport à la sémantique opérationnelle, garantissant que les termes bien typés du langage pourront être exécutés sans erreurs (liées aux types). De plus les algorithmes permettant une utilisation réelle d'un tel système ont été proposés et caractérisés comme (i) corrects et complets par rapport au système logique (ii) terminant dans tous les cas. L'aspect probablement le plus original réside dans la propagation des types minimaux, qui permet de faire apparaître un certain nombre de propriétés opérationnelles au niveau du contrôle de types. D'autre part, les types minimaux Circus peuvent être inférés automatiquement, ce qui laisse entrevoir une bonne souplesse d'utilisation, tout en laissant tout de même la possibilité au programmeur de typer explicitement ses termes, soit pour vérifier ses assertions de typage, soit pour généraliser ses définitions par sur-typage [1]. Dans les chapitres suivants, dédiés à l'extension progressive du langage, nos types énumérés révèleront un potentiel surprenant en capturant nombre de propriétés liées à l'exécution, et généralement traités (dans le meilleur des cas) par des analyseurs statiques en dehors du contrôle de types.

[1] Il ne s'agit toutefois pas du même genre d'inférence de type que celui de *ML* par exemple. Ici, les variables ont toutes un type assigné par le contexte

Actions impératives et machines abstraites polymorphes

5.1 Introduction

Ce chapitre se propose d'étendre le noyau fonctionnel de *Circus*, en introduisant d'abord un nombre limité d'actions impératives, puis une action particulière, la règle de filtrage, permettant d'appliquer des filtres sur des termes sujets (uniquement des types de base à ce niveau de la présentation). L'étape suivante généralise les règles en permettant le filtrage d'une mémoire de coordination inspirée du modèle *Linda*. Les primitives d'accès associatif à cette mémoire permettent de spécifier d'une manière simple des scénarios de coordination riches, tout en étant bien intégrés au modèle de programmation de *Circus*.

Mais pourquoi introduire des opérations de filtrage et de coordination dans le langage formel que nous nous proposons d'étudier ? Les première vont ouvrir de très riches perspectives dans le traitement des structures de données, ou un langage spécialisé dans la transformation de structures se doit d'exceller : non seulement ce dernier doit offrir une expressivité bien adaptée à la modélisation des structures de données mais aussi à leur accès, tant dans leur forme que leur contenu. Les secondes sont une réponse à la complexité temporelle des schémas d'interaction homme-machine que nous proposons de prendre en compte dans le cadre de cette étude, comme évoqué dans la première partie de ce mémoire.

Après avoir introduit des primitives de coordination, il semble pour le moins indispensable de proposer des primitives pour exprimer des calculs parallèles permettant d'exploiter le modèle de coordination. Le modèle de concurrence présenté est minimal, en ce sens qu'il permet uniquement le lancement synchrone de plusieurs activités partageant le même contexte d'exécution. Toutefois, d'après les données expérimentales acquises, il devrait offrir l'expressivité suffisante pour les applications ciblées.

Enfin, après avoir introduit la notion de systèmes d'actions, simples collections étiquetées et ordonnées de règles, ce chapitre termine sur l'élément central du langage *Circus*, les machines abstraites polymorphes. Ces dernières sont une abstraction d'exécution proche des lambda fonction, mais dont les opérations internes sont de nature impérative. Elles ont pour vocation la composition syntaxique au moyen d'opérateurs dédiés, qui feront l'objets du chapitre suivant.

5.2 Actions impératives

Cette section introduit les éléments de base ouvrant la voie à des traitements impératifs, c'est à dire : (i) basés sur la déclaration de variables à portée locale, (ii) l'affectation, (iii) la composition d'instructions en séquence et (iv) l'itération des traitements. Dans l'approche purement fonctionnelle, les variables sont définies comme paramètres, et leurs portées sont limitées au corps de la fonction, la seule composition est l'appel de fonction avec évaluation des paramètres, et l'itération est remplacée par des appels récursifs. Nous avions évoqué dans l'état de l'art, que ce modèle d'exécution possédait certes ses propres qualitées, mais imposait également un "style" dans la résolution des problèmes. Notamment, la récursion est naturellement associée à un modèle arborescent des structures de données. Nous n'introduisons pas seulement ces aspects impératifs pour élargir (et enrichir) le modèle d'exécution, mais aussi pour converger, à la fin de ce chapitre sur une abstraction de contrôle adaptée à nos objectifs (expressivité orientée vers la transformation de structures, interactive ou non, et favorisant à la fois la composition et la réutilisation).

5.2.1 Syntaxe

La déclaration des variables se fait explicitement en donnant un identificateur (non utilisés dans l'environnement), un type et une valeur. La variable est utilisable en écriture et lecture dans le sous-terme droit (noté e_2 dans la definition syntaxique ci-dessous). L'affectation, "classique", associe un terme à une variable désignée par son identificateur. L'ancienne valeur est perdue, remplacée par la forme réduite du terme. La composition d'instructions en séquence est elle aussi "classique". L'itération est plus originale, car elle ne fait pas apparaitre de condition booléenne, telle qu'on les rencontre dans les constructions *while* ou *repeat*. On ne lui associe pas non plus d'instruction *break*, comme souvent dans les schémas *repeat* $\{$*... if (cond) then break ...*$\}$. Elle utilise le fait qu'une instruction impérative est elle-même évaluée comme un terme de type *Unit*, avec comme forme réduite **none** ou **unit**. Les valeurs **none** correspondent à la fin de l'itération, comme décrit de manière plus formelle dans la sémantique opérationnelle. L'interêt est de garder une seule forme, simple, de l'itération, bien intégrée à la sémantique globale du langage. Ainsi, $*(h \Rightarrow e)$ correspond à appliquer la règle $h \Rightarrow e$ tant qu'elle reste applicable, c'est à dire que le sous-terme h peut être évalué à **true**, et que e est évalué à **unit**. De même, $*(h_1 \Rightarrow (h_2 \Rightarrow e))$ équivaut à appliquer deux règles en cascade, tant que cela est possible.

La syntaxe de ces nouvelles constructions est formellement décrite par les extensions grammaticales suivantes

e	::=	**var** $x : t = e_1 . e_2$	*déclaration de variable*
e	::=	$x = e$	*affectation*
e	::=	$e_1 ; e_2$	*séquence*
e	::=	$*(e)$	*itération*

Pour définir avec précision la sémantique des termes introduisant des variables locale, nous utilisons un terme "interne", caché au programmeur, tout comme l'est le type **unit**. Ce terme sera par ailleurs réutilisé pour définir la sémantique des machines abstraites (plus particulierement leur invocation) dans la section 5.7. La signification informelle de la création de contexte notée $\phi\langle x , e_1 \rangle . e_2$ est d'enrichir le contexte d'exécution avec une nouvelle entrée, désignée par un identificateur x, et initialisée avec une valeur explicite e_1. L'évaluation du sous-terme e_2 peut alors se faire dans ce nouvel environnement (la variable peut être lue, ou même écrite si e_2 est du type **Unit**).

e	::=	$\phi\langle x , e_1 \rangle . e_2$	*création de contexte*

5.2.2 Typage

Le typage des nouveaux termes utilise une fois encore les particularités de nos types énumérés. [te-aff] montre que l'affectation est une opération dont l'évaluation retourne toujours **unit**. [te-iter], qui concerne la seule construction itérative de *Circus*, montre que l'itération $*e$ ne peut recevoir de type correct que si e a pour type minimal **Unit**. Cela signifie par exemple que $*(x = 10)$ n'est pas une expression correctement typée. En effet, le système de type "capture" élégamment le fait que ce calcul ne termine jamais. Il en va de même pour la composition séquentielle, où par exemple $*(x = 10); y = 20$ ne peut pas recevoir de type correct pour la même raison.

Afin de clarifier la manipulation des types **Unit**, nous proposons l'axiome suivant qui exprime simplement le fait que **Unit** est un type ayant un domaine de valuation fini, tout comme nous l'avions fait pour **bool**.

Axiome 5.1 *Egalité structurelle concernant* **Unit**

$$\emptyset \rhd \textbf{Unit in } \{\textbf{unit}, \textbf{none}\} \doteq \textbf{Unit}$$

A présent, nous pouvons présenter les équations principales.

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd e_1 : t_1 \quad \gamma, x : t_1 \rhd e_2 : t_2}{\gamma \rhd \phi\langle x, e_1\rangle . e_2 : t_2} \text{ [te-context]}$$

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd e_1 : t_1 \quad \gamma, x : t_1 \rhd e_2 : t}{\gamma \rhd \textbf{var } x : t_1 = e_1 . e_2 : t} \text{ [te-var]}$$

$$\frac{\gamma, x : t \rhd e : t}{\gamma, x : t \rhd x = e \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-aff]}$$

$$\frac{\gamma \rhd e \preccurlyeq: \textbf{Unit}}{\gamma \rhd *e \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \text{ [te-iter]}$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}}{\gamma \rhd e_1 ; e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-seq1]}$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 ; e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \text{ [te-seq2]}$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd e_1 ; e_2 \preccurlyeq: \textbf{Unit}} \text{ [te-seq3]}$$

5.2.3 Sémantique

$$\frac{[\mathcal{S} \vdash e_1] \to [\mathcal{S} \vdash e_1']}{[\mathcal{S} \vdash \phi\langle x, e_1\rangle.e_2] \to [\mathcal{S} \vdash \phi\langle x, e_1'\rangle.e_2]} \text{ [e-context1]}$$

$$\frac{[\mathcal{S}, x = v \vdash e] \to [\mathcal{S}, x = v' \vdash e']}{[\mathcal{S} \vdash \phi\langle x, v\rangle.e] \to [\mathcal{S} \vdash \phi\langle x, v'\rangle.e']} \text{ [e-context2]}$$

$$[\mathcal{S} \vdash \phi\langle x, v_1\rangle.v_2] \to [\mathcal{S} \vdash v_2] \text{ [e-context3]}$$

$$[\mathcal{S} \vdash \phi\langle x, \textbf{error}\rangle.e] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-context-err1]}$$

$$[\mathcal{S} \vdash \phi\langle x, v\rangle.\textbf{error}] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-context-err2]}$$

$$\frac{[\mathcal{S} \vdash e] \to [\mathcal{S} \vdash e']}{[\mathcal{S} \vdash x = e] \to [\mathcal{S} \vdash x = e']} \text{ [e-aff1]}$$

$$[\mathcal{S}, x = v_1 \vdash x = v_2] \to [\mathcal{S}, x = v_2 \vdash \textbf{unit}] \text{ [e-aff2]}$$

$$[\mathcal{S} \vdash x = \textbf{error}] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-aff-err]}$$

$$[\mathcal{S} \vdash \textbf{var } x : t = e_1.e_2] \to [\mathcal{S} \vdash \phi\langle x, e_1\rangle.e_2] \text{ [e-var]}$$

$$\frac{[\mathcal{S} \vdash e_1] \to [\mathcal{S}' \vdash e_1']}{[\mathcal{S} \vdash e_1; e_2] \to [\mathcal{S}' \vdash e_1'; e_2]} \text{ [e-seq1]}$$

$$[\mathcal{S} \vdash v; e_2] \to [\mathcal{S} \vdash e_2] \text{ [e-seq2]}$$

$$[\mathcal{S} \vdash \textbf{error}; e_2] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-seq-err]}$$

$$[\mathcal{S} \vdash *e] \to [\mathcal{S} \vdash \textbf{if } (e == \textbf{unit}) \textbf{ then } *e \textbf{ else none}] \text{ [e-iter]}$$

5.2.4 Correction du système de types

Le noyau du langage, présenté au chapitre précédant, est muni d'un système de type correct par rapport à la sémantique opérationelle. Cette sous-section se propose de montrer que cette propriété reste valide pour les nouveaux termes impératifs, en procédant cas par cas.

Proposition 5.2 *Préservation du type d'un terme* $\phi\langle x, e_1\rangle.e_2$

$$\gamma \rhd \phi\langle x, e_1\rangle.e_2 : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash \phi\langle x, e_1\rangle.e_2] \to [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve :L'expression $\phi\langle x, e_1\rangle.e_2$ peut prendre tous les types pris par e_2 (cf [te-context]). La preuve utilise l'induction sur la structure du terme, e_1 et e_2 étant supposés vérifier la propriété. L'hypothèse

$\gamma \rhd \phi\langle x, e_1\rangle.e_2 : t$ peut donc être étendue avec $\gamma \rhd e_1 : t_1$, $x \notin \text{dom}(\gamma)$ et $\gamma, x : t_1 \rhd e_2 : t$, car [te-context] est la seule règle définissant le typage du terme considéré. On procède en enumérant les différentes transitions possibles

Cas [e-context1]. Le sous-terme e_1 n'est pas normalisé. Par définition, le contexte d'exécution est laissé invariant. Or e_1 vérifie la propriété de préservation du type (hypothèse d'induction).

Cas [e-context2]. Par définition, le contexte d'exécution est

laissé invariant. Cependant, e_2 vérifie la propriété de préservation du type (hypothèse d'induction), mais il faut montrer que $\Vdash \mathcal{S}, x = v : \gamma, x : t_1$. L'hypothèse nous permet d'affirmer que $\Vdash \mathcal{S} : \gamma$, or nous avons $x \notin dom(\gamma)$, donc $\mathcal{S}, x = v$ reste bien formé (tous les labels doivent être uniques). D'autre part, nous avons $\gamma \rhd v : t_1$ car $\gamma \rhd e_1 : t_1$ et e_1 vérifie la propriété. Donc, par définition de $\Vdash$: (cf définition 4.41),

$$x = v \in \mathcal{S}, x = v \quad \Rightarrow \quad \exists t_1 \text{ tel que } \gamma, x : t_1 \rhd x : t_1 \text{ et } \gamma, x : t_1 \rhd v : t_1$$

et ainsi, $\Vdash \mathcal{S}, x = v : \gamma, x : t_1$ est démontrée.

Cas [e-context3]. Par définition, le contexte d'exécution est invariant. De plus, si

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd v_1 : t_1 \quad \gamma, x : t_1 \rhd v_2 : t_2}{\gamma \rhd \phi\langle x, v_1\rangle.v_2 : t_2} \text{ [te-context]}$$

alors directement $\gamma \rhd v_2 : t_2$ car le terme est fermé et donc indépendant du contexte de typage.

□

Proposition 5.3 *Préservation du type d'un terme* $x = e$

$$\gamma \rhd x = e : t \quad \Rightarrow \quad \begin{cases} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash x = e] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{cases}$$

Preuve : Le terme $x = e$ peut prendre le type **Unit in** $\{$**none**$\}$ (par [te-aff][te-env4]), **Unit** (par te-aff][te-env4][st-sub]) et enfin $\top$ (par te-aff][te-env4][st-sub]). Nous ne developperons que le premier cas. Nous avons donc comme hypothèse,

$$\frac{\dfrac{\gamma', x : t_1 \rhd e : t_1}{\gamma', x : t_1 \rhd x = e \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}} \text{ [te-aff]}}{\gamma \rhd x = e : \textbf{Unit in } \{\textbf{none}\}} \text{ [te-env4]}$$

a présent nous envisageons toutes les transitions possibles sauf [e-aff-err], qui n'est pas applicable puisque **error** n'est pas typable.

cas [e-aff1] Par définition de la transition, le contexte reste constant. D'autre part, par hypothèse d'induction,

$$\gamma \rhd e : t_1 \quad \Rightarrow \quad \gamma \rhd e' : t_1$$

cas [e-aff2] Nous avons $\gamma \rhd v_2 : t_1$ par hypothèse. La modification de contexte $\mathcal{S}, x = v_1 \rightarrow \mathcal{S}, x = v_2$ laisse donc intacte la relation de conformité avec γ, car le type de x ne change pas. D'autre part, nous savons par axiome que $\gamma \rhd$ **unit** : **Unit**.

□

Proposition 5.4 *Préservation du type d'un terme* **var** $x : t = e_1 . e_2$

$$\gamma \rhd \textbf{var}\ x : t = e_1 . e_2 : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash \textbf{var}\ x : t = e_1 . e_2] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : Sans difficulté. Utilise une seule équation de typage [te-var], mise en correspondance avec [te-context], et une seule transition qui laisse le contexte invariant ([e-var].

□

Proposition 5.5 *Préservation du type d'un terme itération* $*e$

$$\gamma \rhd *e : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash *e] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : Les terme $*e$ peuvent prendre les types **Unit in** $\{$**none**$\}$ ([te-iter][te-env4]), et par subsomption, **Unit** ([te-iter][te-env4][st-sub]) et enfin $\top$ ([te-iter][te-env4][st-sub]). Nous ne développerons que le premier cas. Les hypothèse liées au typage de $*e$ sont données de manière unique par le dernier numérateur de

$$\dfrac{\dfrac{\gamma \rhd e \preccurlyeq : \textbf{Unit}}{\gamma \rhd *e \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-iter}]}{\gamma \rhd *e : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-env4}]$$

Nous avons une seule transition à considérer, [e-iter], qui laisse le contexte d'exécution invariant. Il nous reste donc à montrer que (avec $\gamma \rhd \textbf{Unit in } \{\textbf{none}\} \doteq \textbf{None}$)

$$\gamma \rhd \textbf{if } e == \textbf{unit then } *e \textbf{ else none} : \textbf{None}$$

Nous pouvons appliquer [te-if] pour obtenir la preuve suivante :

$$\dfrac{\dfrac{\dfrac{\gamma \rhd e \preccurlyeq : \textbf{Unit}}{\gamma \rhd e : \textbf{Unit}}\ [\text{te-env4}] \quad \dfrac{}{\gamma \rhd \textbf{unit} : \textbf{Unit}}}{\gamma \rhd e == \textbf{unit} : \textbf{bool}}\ [\text{te-eq}] \quad \gamma \rhd *e : \textbf{None} \quad \dfrac{}{\gamma \rhd \textbf{none} : \textbf{None}}}{\gamma \rhd \textbf{if } e == \textbf{unit then } *e \textbf{ else none} : \textbf{None}}$$

□

Proposition 5.6 *Préservation du type d'un terme séquence* $e_1 ; e_2$

$$\gamma \rhd e_1 ; e_2 : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash e_1 ; e_2] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : Nous ne considererons que les cas correspondants aux types minimaux $\gamma \rhd e_1 ; e_2 \preccurlyeq :$ **Unit in** $\{$**none**$\}$, $\gamma \rhd e_1 ; e_2 \preccurlyeq :$ **Unit in** $\{$**unit**$\}$, et $\gamma \rhd e_1 ; e_2 \preccurlyeq :$ **Unit**.

cas $\gamma \rhd e_1 ; e_2$: **Unit in** $\{$**none**$\}$ Nous avons comme hypothèse les feuilles de l'arbre suivant (te-seq2 est la seule règle qui permet de typer $e_1 ; e_2$ selon le cas envisagé) :

$$\dfrac{\dfrac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 ; e_2 \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-seq2}]}{\gamma \rhd e_1 ; e_2 : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-env4}]$$

Nous considerons les deux transitions [e-seq1] et [e-seq2].

e-seq1 Par hypothèse d'induction, comme nous avons $\gamma \rhd e_1$: **Unit**, alors $\Vdash \mathcal{S}' : \gamma$ et $\gamma \rhd e_1'$: **Unit**. Par conséquent,

$$\dfrac{\dfrac{\gamma \rhd e_1' : \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 ; e_2 \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-seq2}]}{\gamma \rhd e_1' ; e_2 : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-env4}]$$

e-seq2 Ce cas est plus délicat, car la transition modifie la forme syntaxique du terme. Toutefois, l'environnement d'exécution est laissé inchangé ; il reste donc a prouver que le terme $\gamma \rhd e_2$: **Unit in** $\{$**none**$\}$. Nous avons les hypothès en developpant l'unique règle de typage concernée

$$\dfrac{\dfrac{\gamma \rhd v : \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd v ; e_2 \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-seq2}]}{\gamma \rhd v ; e_2 : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-env4}]$$

Le résultat est deja dans les hypothèses.

cas [$\gamma \rhd e_1 ; e_2$: **Unit in** $\{$**unit**$\}$. Similaire en tout point au cas précédant.

cas [$\gamma \rhd e_1 ; e_2$: **Unit** . Similaire en tout point aux cas précédants.

□

5.3 Filtrage

Cette section présente un opérateur de filtrage visant à offrir des fonctionnalités de réécriture, tout en restant bien intégré au système de types. Le filtrage est une opération booléene tentant d'appliquer un filtre f à un terme sujet e. Si l'opération, notée $e\sharp f$, réussie, alors certaines variables distinguées de f voient leur valeur modifiées. Il est alors possible de poursuivre le traitement dans ce nouveau contexte. Le filtrage peut être vu à la fois comme un test sur la structure d'une donnée, combiné avec des affectations susceptibles de modifier le contexte d'interpretation. Cette richesse sémantique rend l'opération interessante, mais requiert la résolution de certaines anomalies de typage. L'expression $?x \boxplus \%\text{“}s\text{“} \boxplus ?y$ définit un filtre s'appliquant sur des termes de type *chaîne de caractère* (en supposant que les variables libres x, y possèdent ce type dans le contexte considéré). A l'issue d'un appariement réussi avec une chaine v, x et y devront contenir les valeurs telles que $v == x + \text{“}s\text{“} + y$. Toutefois, les opérations implicitement réalisées sur les sous-filtres sont hétérogènes : x et y subiront l'équivalent sémantique de $x = \cdots, y = \cdots$ (affectations) alors que $\%\text{“}s\text{“}$ subi un test $\cdots == \text{“}s\text{“}$ (comparaisons). Du point de vue du typage, ces deux opérations sont différentes. Dans le premier cas, l'opérande affecté à la variables doit avoir un type plus petit ou égal à celui de la variable ([te-aff]). Dans le second cas, il suffit de trouver un type commun (qui peut être $\top$; ce qui correspond à une contrainte minimale sur les opérandes, celle d'avoir un type bien formé).

5.3.1 Syntaxe

Les extensions grammaticales suivantes font apparaitre deux nouvelles catégories syntaxiques : h (“*header*”) désigne la partie conditionnelle d'une règle $h \Rightarrow e$, et f désigne les filtres associés à l'opération de filtrage $e\sharp f$.

e	::=	$h \Rightarrow e$	*règle avec condition*
h	::=	$e\sharp f$	*filtrage d'un terme e par un filtre f*
f	::=	$?x$	*variable substituable*
f	::=	$?$	*substitution quelconque*
f	::=	$\%e$	*terme non substituable*
f	::=	$f_1 \boxplus f_2$	*concatenation de filtres*

La concatenation de filtres, notée $\boxplus$, permet de faire apparaitre des contraintes structurelle, telle que celles qui portent sur la succession de sous-chaînes dans une chaîne sujet. Par exemple un sujet $'unechaine'$ peut etre filtré avec succés par $\%'une' \boxplus ?x$; dans le contexte x aura alors pour valeur $'chaine'$. Cet opérateur s'appliquera egalement par la suite aux sujets de type séquence, ensemble et dictionnaire, avec une sémantique similaire, en relation directe avec la sémantique de $+$ pour les types considérés : l'invariant peut s'écrire de manière “pseudo-formelle”

$$\text{si } e\sharp f_1 \boxplus f_2 \quad \text{alors} \quad \exists e_1, e_2 \text{ tels que } e = e_1 + e_2 \text{ avec } e_1\sharp f_1 \text{ et } e_2\sharp f_2.$$

5.3.2 Règles de typage

Nous présentons d'abord le typage des règles, qui a pour but de propager les informations liées à la terminaison (en plus de la fonction première qui est de s'assurer de la compatibilité du type des expressions). Contrairement à la démarche suivie jusqu'alors, nous définissons directement le typage minimal au lieu de le prouver en raisonnant sur la définition [te-min]. Le but est de restreindre les developpements de ce chapitre à l'éssentiel. Il est facile de montrer que ces définitions restent correctes, car la sémantique

opérationnelle des règles montre qu'elles se réécrivent sous forme de tests dont le typage minimal à été étudié en détail.

$$\frac{\gamma \rhd h : \textbf{bool in } \{\textbf{false}, \textbf{none}\} \quad \gamma \rhd e : \textbf{Unit}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \text{ [te-rule]}$$

$$\frac{\gamma \rhd h : \textbf{bool in } \{\textbf{true}\} \quad \gamma \rhd e : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \text{ [te-rule2]}$$

$$\frac{\gamma \rhd h : \textbf{bool in } \{\textbf{true}\} \quad \gamma \rhd e : \textbf{Unit in } \{\textbf{unit}\}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-rule3]}$$

$$\frac{\gamma \rhd h : \textbf{bool in } \{\textbf{true}\} \quad \gamma \rhd e \preccurlyeq: \textbf{Unit}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit}} \text{ [te-rule4]}$$

$$\frac{\gamma \rhd h \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e : \textbf{Unit}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit}} \text{ [te-rule5]}$$

$$\frac{\gamma \rhd h \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{none}\} \quad \gamma \rhd e : \textbf{Unit}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit}} \text{ [te-rule6]}$$

$$\frac{\gamma \rhd h \preccurlyeq: \textbf{bool} \quad \gamma \rhd e : \textbf{Unit}}{\gamma \rhd h \Rightarrow e \preccurlyeq: \textbf{Unit}} \text{ [te-rule7]}$$

Les règles suivantes comportent certaines subtilités qu'il est nécéssaire de mettre en évidence. Ainsi, [te-cat], exprime une contrainte sur le typage des filtres construits à l'aide de $\boxplus$. Ces derniers ne peuvent pas être des types **string** énumérés. En effet, il serait impossible de garantir la préservation des types lors de l'exécution d'un filtrage tel que

$$x : \textbf{string in } \{\text{“}a''\} \rhd \text{“une chaine”} \sharp (\text{ “une”} \quad \boxplus \quad ?x)$$

, qui affecterait la chaîne “ chaine” à la variable x, mettant alors en défaut la cohérence de type. Bien sur, il serait possible d'ajouter des conditions de filtrage implicites pour prévenir de telles incohérences, au prix de performances réduites. C'est donc un choix de conception d'interdire de telles expressions au moyen de conditions restrictives sur les types ([te-invar] et [te-match-free]). Enfin, [te-match-free] et [te-match-free2] indiquent que le résultat d'un filtrage avec $?$ est un booléen ne pouvant jamais être **none** ou **false** (un tel filtrage réussit toujours). De plus, [te-match] interdit d'utiliser le filtrage avec des opérandes de types hétérogènes (n'ayant pas de super-type en commun autre que $\top$), comme par exemple $10 \sharp \text{“}s''$. Ici encore, c'est un choix lié à des aspects de performance, mais aussi opérationnels. La supériorité du filtrage sur l'égalité est que ce premier permet de faire apparaitre explicitement le contexte grâce aux substitutions de variables. Ces substitutions n'ont de sens que si les opérandes possèdent des types compatibles.

$$\frac{\gamma \rhd e \preccurlyeq: \textbf{string} \quad \gamma \rhd e\sharp f_1 : \textbf{bool} \quad \gamma \rhd e\sharp f_2 : \textbf{bool}}{\gamma \rhd e\sharp f_1 \boxplus f_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-cat]}$$

$$\frac{\gamma, x : t \rhd e : t}{\gamma, x : t \rhd e\sharp ?x \preccurlyeq: \textbf{bool in } \{\textbf{true}\}} \text{ [te-match-free]}$$

$$\frac{\gamma \rhd e : t}{\gamma \rhd e\sharp ? \preccurlyeq: \textbf{bool in } \{\textbf{true}\}} \text{ [te-match-free2]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd t_1 \approx t_2}{\gamma \rhd e_1\sharp\%e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-invar]}$$

$$\frac{\gamma \rhd e_1 : \top \textbf{ in } e \quad \gamma \rhd e_2 : \top \textbf{ in } e}{\gamma \rhd e_1\sharp\%e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}\}} \text{ [te-invar-true]}$$

5.3.3 Sémantique

Nous écrivons f_i les termes décrits par les règles de grammaire du type $f ::= \ldots$, et $\boldsymbol{f}$ les termes de la même catégorie syntaxique, mais sous forme réduite (ou normale). Les premières règles [e-match3] et [e-match4] montrent que l'opération de filtrage commence d'abord par normaliser le sous-terme gauche, puis le droit. C'est donc sur des opérandes réduites à leurs formes normales que s'exécute le filtrage proprement dit.

$$[\mathcal{S} \vdash h \Rightarrow e] \to [\mathcal{S} \vdash \textbf{if } h \textbf{ then } e \textbf{ else none}] \text{ [e-rule]}$$

$$\frac{[\mathcal{S} \vdash e] \to [\mathcal{S} \vdash e']}{[\mathcal{S} \vdash e\sharp f] \to [\mathcal{S} \vdash e'\sharp f]} \text{ [e-match3]} \qquad \frac{[\mathcal{S} \vdash f] \to [\mathcal{S} \vdash f']}{[\mathcal{S} \vdash v\sharp f] \to [\mathcal{S} \vdash v\sharp f']} \text{ [e-match4]}$$

Les équations [e-cat2] et [e-cat3] définissent la réduction des filtres construits à l'aide de l'opérateur $\boxplus$, comme étant prioritaire à gauche. Les filtres $?x$ et ? sont eux-mêmes sous forme normalisée ([e-free] et [e-free2]), et les filtres constants sont normalisés quand l'expression incluse est elle-même normalisée ([e-const] et [e-const2]).

$$\frac{[\mathcal{S} \vdash f_1] \to [\mathcal{S} \vdash f_1']}{[\mathcal{S} \vdash f_1 \boxplus f_2] \to [\mathcal{S} \vdash f_1' \boxplus f_2]} \text{ [e-cat2]} \qquad \frac{[\mathcal{S} \vdash f_2] \to [\mathcal{S} \vdash f_2']}{[\mathcal{S} \vdash \boldsymbol{f} \boxplus f_2] \to [\mathcal{S} \vdash \boldsymbol{f} \boxplus f_2']} \text{ [e-cat3]}$$

$$[\mathcal{S} \vdash ?x] \to [\mathcal{S} \vdash ?x] \text{ [e-free]} \qquad [\mathcal{S} \vdash ?] \to [\mathcal{S} \vdash ?] \text{ [e-free2]}$$

$$\frac{[\mathcal{S} \vdash e] \to [\mathcal{S} \vdash e']}{[\mathcal{S} \vdash \%e] \to [\mathcal{S} \vdash \%e']} \text{ [e-const]} \qquad [\mathcal{S} \vdash \%v] \to [\mathcal{S} \vdash \%v] \text{ [e-const2]}$$

Enfin, la dernière série d'équations fixe la sémantique de l'opération de filtrage elle-même. Les filtres ? s'apparient à toute valeure normale v, sans modifier le contexte ([e-match-free2]). Plus riche, [e-match-free] indique que la variable x référencée dans un filtre $?x$ est initalisée après appariement (modification du contexte).

$$[\mathcal{S}, x = v' \vdash v\sharp?x] \rightarrow [\mathcal{S}, x = v \vdash \textbf{true}] \quad \text{[e-match-free]}$$

$$[\mathcal{S} \vdash v\sharp?] \rightarrow [\mathcal{S} \vdash \textbf{true}] \quad \text{[e-match-free2]}$$

$$[\mathcal{S} \vdash \textbf{error}\sharp?x] \rightarrow [\mathcal{S} \vdash \textbf{error}] \quad \text{[e-match-err]}$$

Un terme normal v_1 s'apparie avec un filtre constant $\%v_2$ si ils sont égaux au sens défini par l'opérateur $==$ du langage (cf [e-match] ; ici non plus, le contexte n'est pas modifié.

$$\frac{[\mathcal{S} \vdash v_1 == v_2] \twoheadrightarrow\circ \ [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash v_1\sharp\%v_2] \rightarrow [\mathcal{S} \vdash \textbf{true}]} \ \text{[e-match1]}$$

$$\frac{[\mathcal{S} \vdash v_1 == v_2] \twoheadrightarrow\circ \ [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash v_1\sharp\%v_2] \rightarrow [\mathcal{S} \vdash \textbf{false}]} \ \text{[e-match2]}$$

Enfin, les règles [e-match-cat] et [e-match-cat2], plus complexes, exhibent clairement la nature récursive de l'opération de filtrage : un filtre composite (construit avec $\boxplus$) s'apparie avec un terme si et seulement si ses sous-filtres s'apparient avec des sous-termes correspondants.

il existe v_1, v_2 tels que

$$\frac{[\mathcal{S} \vdash v == v_1 + v_2] \twoheadrightarrow\circ \ [\mathcal{S} \vdash \textbf{true}] \quad [\mathcal{S} \vdash v_1\sharp\boldsymbol{f_1}] \twoheadrightarrow\circ \ [\mathcal{S}' \vdash \textbf{true}] \quad [\mathcal{S}' \vdash v_2\sharp\boldsymbol{f_2}] \twoheadrightarrow\circ \ [\mathcal{S}'' \vdash \textbf{true}]}{[\mathcal{S} \vdash v\sharp\boldsymbol{f_1} \boxplus \boldsymbol{f_2}] \rightarrow [\mathcal{S}'' \vdash \textbf{true}]} \ \text{[e-match-cat]}$$

il n'existe aucun v_1, v_2 tels que

$$\frac{[\mathcal{S} \vdash v == v_1 + v_2] \twoheadrightarrow\circ \ [\mathcal{S} \vdash \textbf{true}] \quad [\mathcal{S} \vdash v_1\sharp\boldsymbol{f_1}] \twoheadrightarrow\circ \ [\mathcal{S}' \vdash \textbf{true}] \quad [\mathcal{S}' \vdash v_2\sharp\boldsymbol{f_2}] \twoheadrightarrow\circ \ [\mathcal{S}'' \vdash \textbf{true}]}{[\mathcal{S} \vdash v\sharp\boldsymbol{f_1} \boxplus \boldsymbol{f_2}] \rightarrow [\mathcal{S}'' \vdash \textbf{false}]} \ \text{[e-match-cat2]}$$

Notons que cette définition est loin de fournir un algorithme pour réaliser l'opération de manière efficace. La littérature est riche de nombreux algorithmes de *pattern matching*. Notre apport sera ici d'intégrer correctement le filtrage dans le système de type (et *a fortiori* de prendre en compte les types de données proposés dans le chapitre suivant.

5.3.4 Correction du système de types

Nous devons a présent établir la correction des deux nouvelles extensions syntaxiques.

Proposition 5.7 *Préservation du type d'une règle* $h \Rightarrow e$

$$\gamma \rhd h \Rightarrow e : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash h \Rightarrow e] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : Nous ne considererons que les types minimaux définies par les règles [te-rule] et [te-rule1] à [te-rule7], combinés avec [te-env4].

cas $\gamma \rhd h \Rightarrow e$: **Unit in** $\{\textbf{none}\}$. Ce typage est obtenu soit par [te-rule][te-env4], soit par [te-rule2][te-env4].

cas [te-rule [te-env4]]. Les hypothèse sont obtenues par

$$\dfrac{\dfrac{\gamma \rhd h : \textbf{bool in } \{\textbf{false}, \textbf{none}\} \quad \gamma \rhd e : \textbf{Unit}}{\gamma \rhd h \Rightarrow e \preccurlyeq : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-rule}]}{\gamma \rhd h \Rightarrow e : \textbf{Unit in } \{\textbf{none}\}}\ [\text{te-env4}]$$

Nous avons de plus quatres hypothèses distinctes à considérer pour le typage de h. Soit h possede directement le type **bool in** $\{$**false**, **none**$\}$, soit il le possède par subsomption de deux manières différentes :

$$\dfrac{\gamma \rhd h : \textbf{bool in } \{\textbf{false}\} \quad \dfrac{\cdots}{\gamma \rhd \textbf{bool in } \{\textbf{false}\} \preccurlyeq \textbf{bool in } \{\textbf{false}, \textbf{none}\}}}{\gamma \rhd h : \textbf{bool in } \{\textbf{false}, \textbf{none}\}}\ [\text{st-sub}]$$

$$\dfrac{\gamma \rhd h : \textbf{bool in } \{\textbf{none}\} \quad \dfrac{\cdots}{\gamma \rhd \textbf{bool in } \{\textbf{none}\} \preccurlyeq \textbf{bool in } \{\textbf{false}, \textbf{none}\}}}{\gamma \rhd h : \textbf{bool in } \{\textbf{false}, \textbf{none}\}}\ [\text{st-sub}]$$

soit par application de [te-env4]

$$\dfrac{\gamma \rhd h \preccurlyeq : \textbf{bool in } \{\textbf{false}, \textbf{none}\}}{\gamma \rhd h : \textbf{bool in } \{\textbf{false}, \textbf{none}\}}\ [\text{te-env4}]$$

La seule transition à examiner est [e-rule], qui laisse le contexte inchangé. Il reste donc à montrer $\gamma \rhd$**if** h **then** e **else none** : **Unit in** $\{$**none**$\}$ ou pour simplifier $\gamma \rhd$**if** h **then** e **else none** : **None** (car $\gamma \rhd$ **None** $\doteq$ **Unit in** $\{$**none**$\}$ et par congruence) pour la première hypothèse sur h et pour la seconde et troisième hypothèse sur h

$$\dfrac{\gamma \rhd h : \textbf{None} \quad \gamma \rhd e : \textbf{Unit} \quad \gamma \rhd \textbf{none} : \textbf{None}}{\gamma \rhd \textbf{if } h \textbf{ then } e \textbf{ else none} : \textbf{None}}\ [\text{te-if-none}]$$

$$\dfrac{\gamma \rhd h : \textbf{bool in } \{\textbf{false}\} \quad \gamma \rhd e : \textbf{Unit} \quad \gamma \rhd \textbf{none} : \textbf{None}}{\gamma \rhd \textbf{if } h \textbf{ then } e \textbf{ else none} : \textbf{None}}\ [\text{te-if-false}]$$

et enfin, le cas le plus simple

$$\dfrac{}{\gamma \rhd \textbf{if } h \textbf{ then } e \textbf{ else none} : \textbf{None}}\ [\text{te-if}]$$

□

Proposition 5.8 *Préservation du type d'une opération de filtrage*

$$\gamma \rhd e \sharp f : t \quad \Rightarrow \quad \begin{cases} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash e \sharp f] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{cases}$$

Preuve :L'expression $e\sharp f$ peut prendre deux types minimaux : **bool in** $\{$**true**, **false**$\}$ ([te-cat],[te-invar]), **bool in** $\{$**true**$\}$ ([te-match-free],[te-match-free2][te-invar-true]). Nous examinerons seulement ces cas, les autre cas étant dérivable simplement par application de [st-sub]. Les preuves utilisent l'induction sur la structure des termes $e\sharp f$.

Cas $\gamma \rhd e\sharp f$: **bool in** $\{$**true**, **false**$\}$.

Il est obtenu par [te-invar] ou [te-cat], combinées avec [te-env4] :

$$\frac{\gamma \rhd e\sharp f \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}}{\gamma \rhd e\sharp f : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-env4]}$$

Nous distinguons deux cas

1. $e_1\sharp\%e_2$

   Les hypothèses associées sont données de manière unique par

   $$\frac{\gamma \rhd e_1 \preccurlyeq: t_1 \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd t_1 \approx t_2}{\gamma \rhd e_1\sharp\%e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-invar]}$$

   À présent, nous examinons les trois dérivations possibles pour le terme considéré

   (a) $[\mathcal{S} \vdash e_1\sharp\%e_2] \to [\mathcal{S} \vdash e_1'\sharp\%e_2]$ [e-match3], avec comme hypothèse unique

   $$[\mathcal{S} \vdash e_1] \to [\mathcal{S} \vdash e_1']$$

   Or la proposition sur la préservation des types 4.44(hypothèse d'induction) nous permet d'affirmer

   $$\gamma \rhd e_1 : t \quad \Rightarrow \quad \gamma \rhd e_1' : t$$

   Or, par [te-env4]

   $$\gamma \rhd e_1 \preccurlyeq: t \quad \Rightarrow \quad \gamma \rhd e_1 : t \quad \Rightarrow \quad \gamma \rhd e_1' : t$$

   Or par definition [te-min], $\gamma \rhd e_1 : t' \Rightarrow \gamma \rhd t \preccurlyeq t'$. Comme nous avons preservation du type pour e_1', $\gamma \rhd e_1' : t' \Rightarrow \gamma \rhd t \preccurlyeq t'$. Finalement,

   $$\gamma \rhd e_1 \preccurlyeq: t \quad \Rightarrow \quad \gamma \rhd e_1' \preccurlyeq: t$$

   par conséquent

   $$\frac{\dfrac{\gamma \rhd e_1' \preccurlyeq: t_1 \quad \gamma \rhd e_2 \preccurlyeq: t_2 \quad \gamma \rhd t_1 \approx t_2}{\gamma \rhd e_1'\sharp\%e_2 \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-invar]}}{\gamma \rhd e_1'\sharp\%e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-env4]}$$

   (b) $[\mathcal{S} \vdash v_1\sharp\%e_2] \to [\mathcal{S} \vdash v_1\sharp\%e_2']$ [e-match4][e-const] avec comme hypothèses

   $$[\mathcal{S} \vdash e_2] \to [\mathcal{S} \vdash e_2']$$

   Nous avons l'hypothèse d'induction qui permet d'affirmer que $\gamma \rhd e_2' \preccurlyeq: t$. D'autre part, $\mathcal{S}$ est laissé inchangé. Donc

   $$\frac{\dfrac{\gamma \rhd e_1 \preccurlyeq: t_1 \quad \gamma \rhd e_2' \preccurlyeq: t_2 \quad \gamma \rhd t_1 \approx t_2}{\gamma \rhd e_1\sharp\%e_2' \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-invar]}}{\gamma \rhd e_1\sharp\%e_2' : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-env4]}$$

(c) $[\mathcal{S} \vdash v_1 \sharp \% v_2] \rightarrow [\mathcal{S} \vdash \textbf{true}]$ [e-match1] avec comme hypothèses

$$[\mathcal{S} \vdash v_1 == v_2] \twoheadrightarrow\!\circ \ \ [\mathcal{S} \vdash \textbf{true}]$$

L'environnement est constant. Il reste à prouver que **true** est bien typé par

$$\frac{\gamma \ \rhd \ \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd \textbf{true} \rightsquigarrow \textbf{true}}{\gamma \rhd \textbf{true} : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \ \text{[te-enum]}$$

(d) $[\mathcal{S} \vdash v_1 \sharp \% v_2] \rightarrow [\mathcal{S} \vdash \textbf{false}]$ [e-match2] Ce cas est similaire au précédent

2. $e_1 \sharp f_1 \boxplus f_2$ Les hypothèses sont données par [te-cat] :

$$\gamma \rhd e_1 \preccurlyeq : \textbf{string}, \quad \gamma \rhd e_1 \sharp f_1 : \textbf{bool}, \quad \gamma \rhd e_1 \sharp f_2 : \textbf{bool}$$

Les dérivations de ce terme sont décrites par [e-match3], [e-match4][e-cat2], [e-match4] [e-cat3] et [e-match-cat]

(a) $[\mathcal{S} \vdash e_1 \sharp f_1 \boxplus f_2] \rightarrow [\mathcal{S} \vdash e_1' \sharp f_1 \boxplus f_2]$ par [e-match3]
Il n'est pas difficile de démontrer la propriété voulue en utilisant l'hypothèse d'induction.

(b) $[\mathcal{S} \vdash v_1 \sharp f_1 \boxplus f_2] \rightarrow [\mathcal{S} \vdash v_1 \sharp f_1' \boxplus f_2]$ par [e-match4][e-cat2]
L'environnement reste inchangé. D'autre part, l'utilisation de [e-cat2] implique que f_1 n'est pas sous forme normale.

$$\frac{\gamma \rhd v_1 \preccurlyeq : \textbf{string} \quad \gamma \rhd v_1 \sharp f_1' : \textbf{bool} \quad \gamma \rhd v_1 \sharp f_2 : \textbf{bool}}{\gamma \rhd v_1 \sharp f_1' \boxplus f_2 \preccurlyeq : \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \ \text{[te-cat]}$$

Or le jugement $\gamma \rhd v_1 \sharp f_1' : \textbf{bool}$ est prouvable en utilisant l'hypothèse d'induction :

$$\gamma \rhd v_1 \sharp f_1 : \textbf{bool} \ \ \Rightarrow \ \ [\mathcal{S} \vdash v_1 \sharp f_1] \rightarrow [\mathcal{S} \vdash v_1 \sharp f_1'], \Vdash \mathcal{S} : \gamma, \gamma \rhd v_1 \sharp f_1' : \textbf{bool}$$

(c) $[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1} \boxplus f_2] \rightarrow [\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1} \boxplus f_2']$ par [e-match4][e-cat3]
La démonstration est similaire au cas précédant.

(d) $[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1} \boxplus \boldsymbol{f_2}] \rightarrow [\mathcal{S} \vdash \textbf{true}]$ par [e-match-cat]
Le jugement $\gamma \rhd \textbf{true} : \textbf{bool in } \{\textbf{true}, \textbf{false}\}$ est simple à démontrer. La propriété $\Vdash \mathcal{S}''$: γ se démontre transitivement en utilisant l'hypothèse d'induction.

Cas $\gamma \rhd e \sharp f : \textbf{bool in } \{\textbf{true}\}$. Il est obtenu par [te-match-free] ,[te-match-free2] ou [te-invar-true], combinées avec [te-env4].

Dans ce cas, les cas où les opérandes ne sont pas normales sont traités comme précédemment. Il reste les transitions [e-match-free], [e-match-free2] et [te-invar-true].

1. Cas $\gamma, x : t \rhd v \sharp ? x : \textbf{bool in } \{\textbf{true}\}$ (cf [te-match-free])
Les hypothèses associées sont

$$\gamma, x : t \rhd v : t$$

Nous avons comme unique transition (par [te-match-free])

$$[\mathcal{S}, x = v' \vdash v \sharp ? x] \rightarrow [\mathcal{S}, x = v \vdash \textbf{true}]$$

Il est trivial de démontrer $\gamma \rhd$ **true** : **bool in** $\{$**true**$\}$. Pour l'environnement, si

$$\Vdash \mathcal{S}, x = v' : \gamma, x : t \quad \Rightarrow \quad \begin{cases} \Vdash \mathcal{S} : \gamma \\ \Vdash x = v' : x : t \end{cases}$$

Or

$$\Vdash x = v' : x : t \quad \Leftrightarrow \emptyset \rhd v' : t$$

(l'équivalence vient de ce que $x = v$ est l'unique élément). Comme nous avons $\gamma \rhd v : t$, alors $\Vdash x = v : x : t$ et par conséquent $\Vdash \mathcal{S}, x = v : \gamma, x : t$.

2. Cas $\gamma \rhd v\sharp?$: **bool in** $\{$**true**$\}$ (cf [te-match-free2]) L'environnement reste constant.Il suffir de montrer que **true** possede le type **bool in** $\{$**true**$\}$
3. Cas $\gamma \rhd v_1 \sharp\% v_2$: **bool in** $\{$**true**$\}$ (cf [te-invar-true]) Les hypothèses associées sont

$$\gamma \rhd v_1 : \top \textbf{ in } e \quad \gamma \rhd v_2 : \top \textbf{ in } e$$

ce qui se developpe, en utilisant [te-enum] et [te-equiv]

$$\gamma \rhd v_1 \sim v_2$$

Ceci permet d'affirmer que, par confluence, $v_1 \equiv v_2$. D'ou l'on peut affirmer que $[\mathcal{S} \vdash v_1 == v_1] \rightarrow [\mathcal{S} \vdash$ **true**$]$ par la définition [e-eq1]. Donc la transition

$$[\mathcal{S} \vdash v_1 \sharp\% v_2] \rightarrow [\mathcal{S} \vdash \textbf{true}]$$

est la seule possible. Or **true** possede le type **bool in** $\{$**true**$\}$

□

5.4 Coordination

Cette section présente les abstractions et primitives de *Circus* dédiées à la coordination de processus. Leur but est de clarifier et simplifier les échanges d'information ainsi que la structure temporelle d'une application composée de plusieurs activitées qui coopérent pour réaliser une tâche. Notre choix s'est porté sur une adaptation d'un modèle à la *Linda* [27, 50], qui nous semble offrir un compromis "puissance expressive"/"efficacité adapté à nos objectifs. Le modèle *Linda* s'appuie d'une part sur une mémoire de coordination et quatre primitives atomiques pour écrire (*out*), lire (*rd*) et retirer (*in*) des informations dans cette mémoire, mais également lancer des processus asynchrones (*eval*) dont le résultat sera stocké en mémoire.

La mémoire elle-même se présente comme un espace de tuples (*tuple space*), c'est à dire un multi-ensemble contenant des informations structurées sous forme de tuples. Les deux primitives *rd* et *in* fournissent un filtre pour spécifier les données devant être consultées (accès associatif à la mémoire). Elles sont toutes les deux "bloquantes", contrairement à *out*.

Ces primitives permettent de modéliser simplement les scenarii de coordination les plus courants (exclusion mutuelle, n-producteurs/m-consomateurs,...). De plus l'accès associatif apporte une isolation fonctionnelle des composants : la mémoire constitue le seul couplage (indirect) entre les processus.

Les sous-sections suivantes proposent une adaptation de la mémoire de coordination et des primitives de consultation et d'écriture au système de types. Notamment des extensions non-bloquantes de *rd* et *in* (renommée *read* et *get*) sont proposées (les primitives originales, bloquantes, sont appelées *bread* et *bget*). D'autre part, les données sont échangées avec leur type associé. L'accès associatif est naturellement mis en rapport avec les opérations de filtrage décrites dans la section précédente. Enfin, il sera montré que les extensions proposées preservent la correction du système de types.

La notion de concurrence, inhérente à la notion de coordination, sera developpée dans la section suivante [1].

5.4.1 Syntaxe

On remarquera que les quatre primitives de consultation correspondent à un non-terminal h, c'est à dire qu'elle ne peuvent être invoquées que dans la partie gauche d'une règle $h \Rightarrow e$.

h	::=	**read** $f : t$	*consultation non bloquante*
		bread $f : t$	*consultation bloquante*
		get $f : t$	*consommation non bloquante*
		bget $f : t$	*consommation bloquante*
e	::=	**put** $e : t$	*dépot dans la mémoire de coordination*

5.4.2 Règles de typage

Les règles [te-read] et [te-get] montrent que les actions de consultation non bloquantes sont vues comme des opérations booléennes pouvant prendre les valeurs **true** (dont la sémantique est "il existe dans la mémoire de coordination une valeur ayant exactement le type requis, et pouvant s'apparier avec le filtre"), ou **false** ("soit il n'existe pas de valeur ayant exactement le type requis, soit l'appariement ne peut pas être satisfait"). Il est plus interessant de voir que les actions bloquantes sont vues comme des opérations booléennes retournant toujours **true**.

[1] Cela peut paraitre étrange, mais nous a sembler permettre une progression plus claire dans la présentation du langage.

$$\forall x \notin \mathrm{dom}(\gamma)$$

$$\frac{\gamma, x:t \ \rhd \ x\sharp f : \mathbf{bool}}{\gamma \rhd (\mathbf{read}\ f:t) \preccurlyeq: \mathbf{bool\ in}\ \{\mathbf{true}, \mathbf{false}\}}\ [\text{te-read}] \qquad \frac{\gamma, x:t \ \rhd \ x\sharp f : \mathbf{bool}}{\gamma \rhd (\mathbf{bread}\ f:t) \preccurlyeq: \mathbf{bool\ in}\ \{\mathbf{true}\}}\ [\text{te-bread}]$$

$$\frac{\gamma, x:t \ \rhd \ x\sharp f : \mathbf{bool}}{\gamma \rhd (\mathbf{get}\ f:t) \preccurlyeq: \mathbf{bool\ in}\ \{\mathbf{true}, \mathbf{false}\}}\ [\text{te-get}] \qquad \frac{\gamma, x:t \ \rhd \ x\sharp f : \mathbf{bool}}{\gamma \rhd (\mathbf{bget}\ f:t) \preccurlyeq: \mathbf{bool\ in}\ \{\mathbf{true}\}}\ [\text{te-bget}]$$

$$\frac{\gamma \rhd e : t}{\gamma \rhd (\mathbf{put}\ e:t) \preccurlyeq: \mathbf{Unit\ in}\ \{\mathbf{unit}\}}\ [\text{te-put}]$$

5.4.3 Sémantique

Nous introduisons formellement à présent la notion de mémoire de coordination, qui est un multi-ensemble (noté $\mathcal{M}$) contenant des valeurs normalisées associées à leurs types, normalisés également. Ces couples sont notés $v:u$. Le système de transition s'enrichi donc d'une nouvelle structure, pour prendre la forme générale

$$\mathcal{M} \ \ [\mathcal{S} \vdash e] \rightarrow \mathcal{M}' \ \ [\mathcal{S}' \vdash e']$$

On peut considérer que les notations utilisées jusqu'a présent sont des abbréviations, oú $\mathcal{M}$ est constant. La notation $\mathcal{M}, v:u$ est une abbréviation de $\mathcal{M} \cup \{(v,u)\}$, où $\cup$ est l'union multi-ensembliste qui ajoute (v,u) à $\mathcal{M}$, augmentant le nombre d'occurence de (v,u) dans $\mathcal{M}$

Dans le premier groupe d'équations, il est montré que les arguments des opérations de consultation sont d'abord évalués (le terme, puis le type associé). Les erreurs sont traitées de la manière habituelle.

$$m \in \{\mathbf{read}, \mathbf{bread}, \mathbf{get}, \mathbf{bget}\}$$

$$\frac{\mathcal{M} \ \ [\mathcal{S} \vdash f] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash f']}{\mathcal{M} \ \ [\mathcal{S} \vdash \mathbf{m}\ f:t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \mathbf{m}\ f':t]}\ [\text{e-mem-a}]$$

$$\frac{\mathcal{M} \ \ [\mathcal{S} \vdash t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash t']}{\mathcal{M} \ \ [\mathcal{S} \vdash \mathbf{m}\ \boldsymbol{f}:t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \mathbf{m}\ \boldsymbol{f}:t']}\ [\text{e-mem-b}]$$

$$[\mathcal{S} \vdash \mathbf{m}\ \mathbf{error}:t] \rightarrow [\mathcal{S} \vdash \mathbf{error}] \ \ [\text{e-mem-err1}]$$

$$[\mathcal{S} \vdash \mathbf{m}\ v:\mathbf{error}] \rightarrow [\mathcal{S} \vdash \mathbf{error}] \ \ [\text{e-mem-err2}]$$

L'opération de lecture asynchrone **read** $\boldsymbol{f}:u$, où le filtre $\boldsymbol{f}$ et le type u sont normaux, est évaluée à **true** si il existe dans la mémoire de coordination un couple (v,u) pouvant s'apparier à $\boldsymbol{f}$ ([e-read1]). Dans ce cas, le contexte $\mathcal{S}$ est (possiblement) modifié. Notons que le type u est *exactement* le même, et non pas seulement compatible par rapport à la relation de sous-typage ou à légalité structurelle $\doteq$ [1].

[1] Cela pourrait être envisagé. Ici encore c'est un choix orienté vers la réalisation ; une telle approche laisse entrevoir une gestion de la mémoire par des fonctions de hachage performantes. L'expérimentation pourra déterminer si un choix plus souple presenterait un interêt réel.

$$\frac{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash v' \sharp \boldsymbol{f}] \rightarrow \mathcal{M}, v' : u \ \ [\mathcal{S}' \vdash \textbf{true}]}{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash \textbf{read } \boldsymbol{f} : u] \rightarrow \mathcal{M}, v' : u \ \ [\mathcal{S}' \vdash \textbf{true}]} \ \text{[e-read1]}$$

$$\frac{\forall v' : u \in \mathcal{M}, \quad \mathcal{M} \ \ [\mathcal{S} \vdash v' \sharp \boldsymbol{f}] \rightarrow \mathcal{M} \ \ [\mathcal{S}' \vdash \textbf{false}])}{\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{read } \boldsymbol{f} : u] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \textbf{false}]} \ \text{[e-read2]}$$

La seule différence que présente **get** $f : t$ avec **read** $f : t$ est dans la gestion de $\mathcal{M}$. En cas d'appariement réussi, le couple (u, v) est cette fois retiré de la mémoire ([e-get1]). Comme pour [e-read2], en cas d'échec, l'environnement $\mathcal{S}$ n'est pas modifié ([e-get2]).

$$\frac{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash v' \sharp \boldsymbol{f}] \rightarrow \mathcal{M}, v' : u \ \ [\mathcal{S}' \vdash \textbf{true}]}{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash \textbf{get } \boldsymbol{f} : u] \rightarrow \mathcal{M} \ \ [\mathcal{S}' \vdash \textbf{true}]} \ \text{[e-get1]}$$

$$\frac{\forall v' : u \in \mathcal{M}, \ \ \mathcal{M} \ \ [\mathcal{S} \vdash v' \sharp \boldsymbol{f}] \rightarrow \mathcal{M} \ \ [\mathcal{S}' \vdash \textbf{false}]}{\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{get } \boldsymbol{f} : u] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \textbf{false}]} \ \text{[e-get2]}$$

Les actions bloquantes sont en tout point similaires à leurs contreparties non-bloquantes, mais sont toujours évaluées à vrai. Sur le plan temporel, cela correspond à une "attente" que les conditions nécessaires soient présentes dans l'environnement d'exécution.

$$\frac{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash v' \sharp \boldsymbol{f}] \rightarrow \mathcal{M}, v' : u \ \ [\mathcal{S}' \vdash \textbf{true}]}{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash \textbf{bread } \boldsymbol{f} : u] \rightarrow \mathcal{M}, v' : u \ \ [\mathcal{S}' \vdash \textbf{true}]} \ \text{[e-bread]}$$

$$\frac{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash v' \sharp \boldsymbol{f}] \rightarrow \mathcal{M} \ \ [\mathcal{S}' \vdash \textbf{true}]}{\mathcal{M}, v' : u \ \ [\mathcal{S} \vdash \textbf{bget } \boldsymbol{f} : u] \rightarrow \mathcal{M} \ \ [\mathcal{S}' \vdash \textbf{true}]} \ \text{[e-bget]}$$

Enfin, l'opération **put** $e : t$ consiste à évaluer e, puis t, et à déposer le couple normalisé dans la mémoire $\mathcal{M}$. La gestion d'erreur est "classique".

$$\frac{\mathcal{M} \ \ [\mathcal{S} \vdash e] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash e']}{\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put } e : t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put } e' : t]} \ \text{[e-put-a]}$$

$$\frac{\mathcal{M} \ \ [\mathcal{S} \vdash t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash t']}{\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put } v : t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put } v : t']} \ \text{[e-put-b]}$$

$$\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put } v : u] \rightarrow \mathcal{M}, v : u \ \ [\mathcal{S} \vdash \textbf{unit}] \ \ \text{[e-put]}$$

$$\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put error} : t] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \textbf{error}] \ \ \text{[e-put-err1]}$$

$$\mathcal{M} \ \ [\mathcal{S} \vdash \textbf{put } v : \textbf{error}] \rightarrow \mathcal{M} \ \ [\mathcal{S} \vdash \textbf{error}] \ \ \text{[e-put-err2]}$$

5.4.4 Correction du système de types

D'une manière similaire à $\mathcal{S}$, la structure $\mathcal{M}$ contient des termes "normaux", associés à des types normaux, donc fermés. Cette structure est correctement typée, notée $\emptyset \triangleright \mathcal{M}$,lorsque tous les couples (terme,type), notés $v : u$ sont bien typés.

Définition 5.9 *Mémoire de coordination bien typée*

$$\emptyset \rhd \mathcal{M} \textit{ ssi } \forall v : u \in \mathcal{M}, \emptyset \rhd v : u$$

La correction du système de types impose que les opérations impliquant la mémoire de coordination $\mathcal{M}$ préservent la cohérence des couples (terme,type) contenus par $\mathcal{M}$.

Proposition 5.10 *Correction du système de types (Extension à la coordination)*

$$\gamma \rhd e : t \quad \Rightarrow \quad \begin{cases} \textit{pour tout } \mathcal{S},\ \mathcal{M} \quad \textit{tels que} \quad \Vdash \mathcal{S} : \gamma, \emptyset \rhd \mathcal{M}, \\ \mathcal{M}\ \ [\mathcal{S} \vdash e] \rightarrow \mathcal{M}'\ \ [\mathcal{S}' \vdash e'] \\ \textit{avec} \quad \gamma \rhd e' : t \quad \textit{et} \quad \Vdash \mathcal{S}' : \gamma, \emptyset \rhd \mathcal{M}' \end{cases}$$

Preuve :(Principe). Considère uniquement les opérations affectant $\mathcal{M}$, en lecture, écriture ou consommation. Les autres actions laissent $\mathcal{M}$ invariant.

cas "put $e : t$" Lorsque cette opération est bien typée, ses réductions conservent cette propriété. Pour le terme e, il suffit d'utiliser la proposition initiale sur la préservation des types 4.44. Pour la réduction du type associé (t), [te-put] et les lemmes 4.45 et 4.15 permettent d'affirmer la propriété équivalente. ainsi, le couple initial $e : t$ devenu après réduction $v : u$ reste cohérent. L'écriture de ce couple dans $\mathcal{M}$ préserve donc la cohérence de $\mathcal{M}$, au sens défini par 5.9. D'autre part, [e-put] montre que la forme réduite de l'opération **put** $e : t$ est **unit**, et $\gamma \rhd$ **unit** : **Unit** pour tout γ.

cas "read $f : t$" Les règles [te-read] et [st-sub] nous montrent que **read** $f : t$ peut avoir comme types **bool in** $\{$**true**, **false**$\}$, **bool** et $\top$. Nous considerons d'abord le premier cas, les autres etant similaires (peuvent être mis en rapport au premier cas par application systèmatique de [st-sub]). Dans ce cas, $\mathcal{M}$ est inchangée. Il faut seulement montrer que $\gamma \rhd$ **read** $f' : t$: **bool in** $\{$**true**, **false**$\}$ et $\Vdash \mathcal{S}' : \gamma$. Si f' n'est pas normal,[te-read] nous montre que $\gamma \rhd f' : t$ est vrai car le numérateur reste vrai ([te-match]).

Si f' est normal, t peut être normal ou non. Dans ces deux cas, le lemme 4.45 permet d'affirmer, avec [te-read], que $\gamma \rhd$ **read** $f' : t$: **bool in** $\{$**true**, **false**$\}$ reste vrai.

Si **read** $f : t$ se réduit à **true**, nous avons trivialement $\gamma \rhd$ **true** : **bool in** $\{$**true**, **false**$\}$. D'autre part, $\mathcal{S}'$ reste cohérent de part [e-read1] et 5.8.

Si **read** $f : t$ se réduit à **false**, alors directement $\gamma \rhd$ **true** : **bool in** $\{$**true**, **false**$\}$, et [te-read2] nous montre que $\mathcal{S}$ reste inchangée ($\mathcal{S}' = \mathcal{S} \ \Rightarrow \Vdash \mathcal{S}' : \gamma$).

Les autres cas (**bread** $f : t$, **get** $f : t$, **bget** $f : t$) sont traités de manière similaire. □

5.5 Concurrence

La notion de concurrence présentée dans cette section se réduit à une seule primitive qui permet de lancer deux évaluations en parallèle, et qui se termine elle-même lorsque les deux évaluations sont terminées. Les sous-processus partagent le même environnement d'exécution, et peuvent se synchroniser et communiquer au moyen de la mémoire de coordination $\mathcal{M}$. Des effets de bord peuvent apparaître, si par exemple il existe des variables communes aux deux processus modifiées en écriture.

5.5.1 syntaxe

$$e \quad ::= \quad e_1 \| e_2 \quad \textit{actions concurrentes}$$

5.5.2 typage

Ici encore, le typage a pour but d'assurer la compatibilité des types, mais aussi la compositionalité par rapport aux itérations

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd e_1 \| e_2 \preccurlyeq: \textbf{Unit}} \text{ [te-par]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 \| e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \text{ [te-par2]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}}{\gamma \rhd e_1 \| e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-par3]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 \| e_2 \preccurlyeq: \textbf{Unit}} \text{ [te-par4]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}}{\gamma \rhd e_1 \| e_2 \preccurlyeq: \textbf{Unit}} \text{ [te-par5]}$$

Ces équations permettent par exemple d'affirmer que l'expression suivante est correctement typée,

$$\emptyset \rhd \textbf{var } x : \textbf{num} = 0 .*(\textbf{none} \| x = x + 1) : \textbf{Unit in } \{\textbf{none}\}$$

Ce qui revient à dire qu'il est possible que cette itération termine. Si l'on remplace la composition parallèle par la composition séquentielle, l'expression ne peut plus être typée. On voit la un double effet : celui de l'indeterminisme lié au parallélisme (un des deux sous-processus termine le premier) ,et des règles sémantiques [e-par-endL] et [e-par-endR] qui fixent la politique de terminaison (non synchronisée).

5.5.3 sémantique

Les règles [e-par-L] et [e-par-R] décrivent une sémantique d'entrelacement (*interleaving*), à la base de nombreuses formalisations de processus concurrents [82, 62, 83, 61, 86]. Mais elles exhibent également la nature indéterministe de l'évolution d'un système concurrent asynchrone (les deux règles sont applicables à tout moment, indistinctement). Les règles [e-par-endL] et [e-par-end-R] montrent que lorsqu'un

des deux sous-processus prend la forme d'une valeur normale (fin de l'évaluation), la concurrence prend fin, et le sous-processus restant devient le processus principal. Enfin, [e-par-errL] et [e-par-errR] montrent qu'une erreur dans un des sous-processus provoque l'interruption des autre sous-processus d'une part, et donne comme résultat une erreur.

$$\frac{\mathcal{M}\ [\mathcal{S} \vdash e_1] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_1']}{\mathcal{M}\ [\mathcal{S} \vdash e_1 \| e_2] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_1' \| e_2]}\ \text{[e-par-L]}$$

$$\frac{\mathcal{M}\ [\mathcal{S} \vdash e_2] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_2']}{\mathcal{M}\ [\mathcal{S} \vdash e_1 \| e_2] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_1 \| e_2']}\ \text{[e-par-R]}$$

$$\mathcal{M}\ [\mathcal{S} \vdash v \| e] \to \mathcal{M}\ [\mathcal{S} \vdash e]\ \text{[e-par-endL]}(v \in \{\textbf{unit}, \textbf{none}\})$$

$$\mathcal{M}\ [\mathcal{S} \vdash e \| v] \to \mathcal{M}\ [\mathcal{S} \vdash e]\ \text{[e-par-endR]}(v \in \{\textbf{unit}, \textbf{none}\})$$

$$\mathcal{M}\ [\mathcal{S} \vdash \textbf{error} \| e] \to \mathcal{M}\ [\mathcal{S} \vdash \textbf{error}]\ \text{[e-par-errL]}$$

$$\mathcal{M}\ [\mathcal{S} \vdash e \| \textbf{error}] \to \mathcal{M}\ [\mathcal{S} \vdash \textbf{error}]\ \text{[e-par-errR]}$$

5.5.4 Correction du système de types

La proposition suivante ne pose pas de problèmes mathématiques particuliers, de part la simplicité des hypothèse d'exécution.

Proposition 5.11 *Préservation du type lors des réductions (extension de 4.44 à la concurrence)*

$$\gamma \rhd e_1 \| e_2 : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout } \mathcal{S}, \mathcal{M} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \textit{ et } \emptyset \rhd \mathcal{M}, \\ \mathcal{M}\ [\mathcal{S} \vdash e_1 \| e_2] \to \mathcal{M}'\ [\mathcal{S}' \vdash e'] \textit{ avec} \\ \emptyset \rhd \mathcal{M}',\ \ \gamma \rhd e' : t,\ \ \textit{et}\ \ \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve :(Esquisse). Par induction sur la structure de $e_1 \| e_2$. Les termes $e_1 \| e_2$ peuvent uniquement prendre les types **Unit in** $\{$**unit**$\}$, **Unit in** $\{$**none**$\}$, **Unit** ([te-par] ou [te-par4] ou [te-par5],[te-env4]) ou $\top$ [st-sub]. Nous developpons uniquement le troisieme cas.

- **cas** $\mathcal{M}\ [\mathcal{S} \vdash e_1 \| e_2] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_1' \| e1]$**.** correspond à l'application de [e-par-L], c'est à dire de son numérateur $\mathcal{M}\ [\mathcal{S} \vdash e_1] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_1']$, et donc nous avons bien $\gamma \rhd e_1$: **Unit** (de part le numérateur de [te-par]), $\Vdash \mathcal{S}' : \gamma$ et enfin $\emptyset \rhd \mathcal{M}'$.
- **cas** $\mathcal{M}\ [\mathcal{S} \vdash e_1 \| e_2] \to \mathcal{M}'\ [\mathcal{S}' \vdash e_1 \| e_2']$**.** Identique au précédant.
- **cas** $\mathcal{M}\ [\mathcal{S} \vdash v \| e_2] \to \mathcal{M}\ [\mathcal{S} \vdash e_2]$**.** Correspond à l'application de [e-par-endL], avec comme condition $v \in \{$**unit**, **none**$\}$. Nous avons encore $\gamma \rhd e_2$: **Unit** (de part le numérateur de [te-par]), et de plus $\mathcal{M}$ et $\mathcal{S}$ sont laissés invariants.
- **cas** $\mathcal{M}\ [\mathcal{S} \vdash e_1 \| v] \to \mathcal{M}\ [\mathcal{S} \vdash e_1]$**.** Identique au précédent.

Les cas correspondants à [e-par-errL] et [e-par-errR] sont éliminés par l'hypothèse d'induction, puisque d'après [te-par], $\gamma \rhd e_1$: **Unit** et $\gamma \rhd e_2$: **Unit** □

5.6 Système d'actions

L'interêt principal des systèmes d'actions que nous proposons à présent est de pouvoir définir simplement une collection de règles. Chacune de ces règles doit être essayée en séquence, selon l'ordre de déclaration. Le fait que ces règles soient étiquetées prend son importance avec les opérations de composition sur les machines abstraites polymorphes présentées plus loin. Nous verrons alors que cette désignation permet de construire de nouvelles machines abstraites par *projection* (des règles sont éliminées, d'autres sont conservées).

5.6.1 Syntaxe

Nous introduisons une nouvelle catégorie syntaxique, notée m, qui sera utilisée dans la prochaine section pour définir la syntaxe des machines abstraites polymorphes. Sans cette précision, son utilisation présente pourrait sembler inutile.

$$\begin{array}{lcll} e & ::= & m & \\ m & ::= & \{\ell_1 : e_1, \cdots, \ell_n : e_n\} & \textit{actions ordonnées et étiquetées} \end{array}$$

5.6.2 Sémantique

Les règles suivantes montrent que du point de vue opérationnel, les systèmes d'actions ne sont autre qu'une collection de tests réalisés dans l'ordre et en cascade. Notons toutefois, que le programmeur ne pourrait pas spécifier de telles expressions directement, car la valeur **unit** n'est pas un terme explicite du langage.

$$[\mathcal{S} \vdash \{\ell_1 : e_1, \cdots, \ell_n : e_n\}] \rightarrow [\mathcal{S} \vdash \begin{array}{l} \textbf{if } (e_1 == \textbf{none}) \\ \quad \textbf{then if } (e_2 == \textbf{none}) \\ \quad\quad \vdots \\ \quad\quad\quad \textbf{then if } (e_n == \textbf{none}) \\ \quad\quad\quad\quad \textbf{then none} \\ \quad\quad\quad\quad \textbf{else unit} \\ \quad\quad\quad \textbf{else unit} \\ \quad\quad \textbf{else unit} \\ \quad \textbf{else unit} \end{array}] \quad \text{[e-asys]}$$

5.6.3 Typage

Les trois premières règles que nous présentons a présent sont assez naturelles à comprendre si on les considère sous l'angle de la sémantique opérationnelle. La quatrième reste plus délicate, car elle couvre un nombre important de typages différents.

pour $\ell_1 \cdots \ell_n \notin \gamma$, et tous distincts

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}}{\gamma \rhd \{\ell_1 : e_1\} \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-asys1a]}$$

$$2 \leq n, 1 < i \leq n \quad \frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_i : \textbf{Unit}}{\gamma \rhd \{\ell_1 : e_1, \cdots, \ell_n : e_n\} \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-asys1]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\} \quad \cdots \quad \gamma \rhd e_n \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd \{\ell_1 : e_1, \cdots, \ell_n : e_n\} \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \text{ [te-asys2]}$$

La prochaine règle, [te-asys] est plus riche que les précédentes, car elle couvre un plus grand nombre de cas. Ainsi, elle peut s'appliquer pour

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit}}{\gamma \rhd \{\ell_1 : e_1\} \preccurlyeq: \textbf{Unit}} \text{ [te-asys]}$$

mais aussi pour

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit} \;\; \gamma \rhd e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\} \;\; \gamma \rhd e_3 \preccurlyeq: \textbf{Unit}}{\gamma \rhd \{\ell_1 : e_1, \ell_2 : e_2, \ell_3 : e_3\} \preccurlyeq: \textbf{Unit}} \text{ [te-asys]}$$

ou encore

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\} \;\; \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd \{\ell_1 : e_1, \ell_2 : e_2\} \preccurlyeq: \textbf{Unit}} \text{ [te-asys]}$$

La règle (générique) est formalisée par

$$\frac{\begin{array}{c} 1 \leq i \leq j \leq n \\ (\gamma \rhd e_i \preccurlyeq: \textbf{Unit} \vee \gamma \rhd e_i \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}) \quad \gamma \rhd e_j \preccurlyeq: \textbf{Unit} \end{array}}{\gamma \rhd \{\ell_1 : e_1, \cdots, \ell_n : e_n\} \preccurlyeq: \textbf{Unit}} \text{ [te-asys]}$$

5.6.4 Correction du système de types

La proposition suivante permet d'étendre la caractérisation du système de types aux systèmes d'action.

Proposition 5.12 *Préservation du type lors des réductions (extension de 4.44 aux systèmes d'action)*

$$\gamma \rhd \{\ell_1 : e_1, \cdots, \ell_n : e_n\} : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout } \mathcal{S}, \mathcal{M} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \textit{ et } \emptyset \rhd \mathcal{M}, \\ \mathcal{M} \;\; [\mathcal{S} \vdash \{\ell_1 : e_1, \cdots, \ell_n : e_n\}] \rightarrow \mathcal{M}' \;\; [\mathcal{S}' \vdash e'] \textit{ avec} \\ \emptyset \rhd \mathcal{M}', \;\; \gamma \rhd e' : t, \;\; \textit{et} \;\; \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve :(Esquisse). Par induction sur la structure de $\{\ell_1 : e_1, \cdots, \ell_n : e_n\}$. Les termes tels que $\{\ell_1 : e_1, \cdots, \ell_n : e_n\}$ peuvent soit prendre les type **Unit** (selon trois arbres, [te-asys][te-env4] ou [te-asys1][st-enum][st-sub] ou encore [te-asys2][st-enum][st-sub]) soit prendre le type $\top$ (par application supplémentaire de [st-sub]) . Nous developpons uniquement les premier cas. Remarquons d'abord que dans tous les cas, l'ensemble des e_i possede le type **Unit**, par subsomption appliquée aux types minimaux $e_i \preccurlyeq:$ **Unit in** $\{$**none**$\}$, $e_i \preccurlyeq:$ **Unit in** $\{$**unit**$\}$ ou par [te-env4] appliquée sur $e_1 \preccurlyeq:$ **Unit**.

La règle [e-asys] réécrit le terme en un arbre de sous-termes **if** $e_1 ==$ **none then** $(\cdots)$ **else unit**, en laissant l'environnement d'exécution constant. Il suffit de montrer que cette dernière expression possède le type **Unit**. Or l'expression **if** $e_n ==$ **none then none else unit** possède le type **Unit** (par [te-if]) :

$$\frac{\dfrac{\gamma \rhd e_n : \textbf{Unit} \;\; \dfrac{\gamma \rhd \textbf{none} : \textbf{None} \;\; \gamma \rhd \textbf{None} \preccurlyeq \textbf{Unit}}{\gamma \rhd \textbf{none} : \textbf{Unit}} \text{ [st-sub]}}{\gamma \rhd e_n == \textbf{none} : \textbf{bool}} \text{ [te-eq]} \;\; \gamma \rhd \textbf{none} : \textbf{Unit} \;\; \dfrac{}{\gamma \rhd \textbf{unit} : \textbf{Unit}}}{\gamma \rhd \textbf{if } e_n == \textbf{none then none else unit} : \textbf{Unit}}$$

Il suffit ensuite de démontrer par récurrence que pour tout i tel que $1 \leq i \leq n$, alors pour tout e

$$\gamma \rhd e : \textbf{Unit} \Rightarrow \gamma \rhd \textbf{if } e_i == \textbf{none then } e \textbf{ else none} : \textbf{Unit}$$

Ce qui est donc vrai en particulier pour e_1 □

5.7 Machines abstraites

Nous appelons *machine abstraite polymorphe*, ou *pam*, une construction qui accepte un unique paramètre en entré et délivre un unique paramètre en résultat. À la différence d'une construction *lambda*, les opérations internes sont de nature impérative, et la forme syntaxique encapsulée dans l'abstraction *pam* est contrainte de manière a garantir des propriétés compositionnelles. En effet, les *pam*s peuvent être utilisées comme des opérandes dans des expressions "algébriques" permettant de construire de nouvelles *pam*s. Le système de type assura la transformation des signatures lors de ces opérations, et garantira la validité de la composition (du point vue du système de type uniquement). Ces points seront développé dans le chapitre 7.

5.7.1 Syntaxe

e	::=	$\nabla x : t_1, y : t_2 . m$	*machines abstraites polymorphes*
m	::=	**var** $x : t . m$	*déclaration de variables locales*

Pour définir avec précision la sémantique de l'invocation des machines abstraites, nous utilisons une fois encore un terme "interne", caché au programmeur. La signification informelle de la création de contexte $\phi\langle x, e_1\rangle.e_2$ (cf section 5.2) est d'enrichir le contexte d'exécution avec une nouvelle entrée, désignée par un identificateur x, et initialisée avec une valeur explicite e_1. L'évaluation du sous-terme e_2 peut alors se faire dans ce nouvel environnement (la variable peut être lue, ou même écrite si e_2 est du type **Unit**).

Le nouveau terme, $\rho\langle x, e_1\rangle.e_2$ ressemble beaucoup à $\phi\langle x, e_1\rangle.e_2$, mais retourne aprés évaluation la valeur de la variable x, et non pas celle de e_2. Cette distinction apparait dans les équations de typage associées (cf [te-eval] ci dessous).

e	::=	$\rho\langle x, e_1\rangle.e_2$	*évaluation dans un contexte*

5.7.2 Typage

Les équation [t-pam] et [te-pam] définissent le type bien formé "machine abstraite" et les termes bien typés leurs correspondant. Ce nouveau type est en tout point semblable au type "fonction" associé aux lambda constructions. Pourtant, il est important de pouvoir distinguer ces deux types lors des opérations de composition qui seront abordées plus tard : en effet, le langage n'offre pas ce genre d'opérateurs pour les fonctions.

$$\frac{\gamma \rhd t_1 \quad \gamma \rhd t_2}{\gamma \rhd t_1 \Longrightarrow t_2} \text{ [t-pam]}$$

$$\frac{\gamma \rhd t_1 \Longrightarrow t_2 \quad x \neq y \quad x, y \notin \text{dom}(\gamma) \quad \gamma, x : t_1, y : t_2 \rhd m : \textbf{Unit}}{\gamma \rhd \nabla x : t_1, y : t_2 . m : t_1 \Longrightarrow t_2} \text{ [te-pam]}$$

À la différence du terme **var** $: = .$ utilisé pour les déclarations de variables locales, le **var** $x : t.m$ associé à la catégorie syntaxique m, et donc aux machines abstraites, est nécessairement de type **Unit**, à la condition nécessaire que le terme inclu m soit du type **Unit**.

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma, x : t \rhd m : \textbf{Unit}}{\gamma \rhd \textbf{var}\ x : t.m : \textbf{Unit}}\ \text{[te-mvar]}$$

$$\frac{\gamma \rhd e_1 : t_1 \Longrightarrow t_2 \quad \gamma \rhd e_2 : t_1}{\gamma \rhd e_1(e_2) : t_2}\ \text{[te-@-pam]}$$

$$\frac{x \notin \text{dom}(\gamma) \quad \gamma \rhd e_1 : t_1 \quad \gamma, x : t_1 \rhd e_2 : \textbf{Unit}}{\gamma \rhd \rho\langle x, e_1\rangle.e_2 : t_1}\ \text{[te-eval]}$$

$$\frac{\gamma \rhd t_2 \preccurlyeq t_2' \quad \gamma \rhd t_1' \preccurlyeq t_1}{\gamma \rhd t_1 \Longrightarrow t_2 \preccurlyeq t_1' \Longrightarrow t_2'}\ \text{[st-pam]}$$

5.7.3 Sémantique

Nous definissons d'abord la sémantique du terme "interne" $\rho\langle x, e_1\rangle.e_2$, car elle est ensuite utilisée pour la définition de l'invocation de machines. [e-eval1] montre qu'il y a priorité dans l'évaluation du terme e_1 initialisant la variable s. Lorsque ce terme est réduit, le terme principal est réduit à son tour [e-eval2], tout en étant susceptible de modifier la valeur associée à x (observer le numérateur, qui fait apparaitre un contexte enrichi d'une entrée $x = v$. Lorsque ce terme principal est lui-même réduit, le terme complet $\rho\langle,\rangle$. se réduit en la valeur associée à x (cf [e-eval3]). C'est cette dernière règle qui marque la différence sémantique d'avec $\phi\langle,\rangle$..

$$\frac{[\mathcal{S} \vdash e_1] \to [\mathcal{S} \vdash e_1']}{[\mathcal{S} \vdash \rho\langle x, e_1\rangle.e_2] \to [\mathcal{S} \vdash \rho\langle x, e_1'\rangle.e_2]}\ \text{[e-eval1]}$$

$$\frac{[\mathcal{S}, x = v \vdash e] \to [\mathcal{S}, x = v' \vdash e']}{[\mathcal{S} \vdash \rho\langle x, v\rangle.e] \to [\mathcal{S} \vdash \rho\langle x, v'\rangle.e']}\ \text{[e-eval2]}$$

$$[\mathcal{S} \vdash \rho\langle x, v_1\rangle.v_2] \to [\mathcal{S} \vdash v_1]\ \text{[e-eval3]}$$

$$[\mathcal{S} \vdash \rho\langle x, \textbf{error}\rangle.e] \to [\mathcal{S} \vdash \textbf{error}]\ \text{[e-eval-err1]}$$

$$[\mathcal{S} \vdash \rho\langle x, v\rangle.\textbf{error}] \to [\mathcal{S} \vdash \textbf{error}]\ \text{[e-eval-err2]}$$

[e-pam] indique qu'une machine abstraite est sous-forme réduite lorsque les types liés aux arguments sont eux-même réduits. La dernière équation [e-@-pam] indique que lors de l'application, le paramètre de retour, sera d'abord initialisé à l'aide d'une fonction interne $\mathcal{I}nit$, puis le paramètre d'entré sera réduit, et enfin le corps principal sera lui-meme réduit dans le contexte ainsi formé.

$$\frac{[\mathcal{S} \vdash t_1] \twoheadrightarrow\!\circ\ [\mathcal{S} \vdash u_1] \quad [\mathcal{S} \vdash t_2] \twoheadrightarrow\!\circ\ [\mathcal{S} \vdash u_2]}{[\mathcal{S} \vdash \nabla x : t_1, y : t_2.m] \to [\mathcal{S} \vdash \nabla x : u_1, y : u_2.m]}\ \text{[e-pam]}$$

$$[\mathcal{S} \vdash \textbf{var}\ x : t.m] \to [\mathcal{S} \vdash \phi\langle x, \mathcal{I}nit(t)\rangle.m]\ \text{[e-mvar]}$$

$$[\mathcal{S} \vdash \nabla x : t_1, y : t_2.m(e_2)] \to [\mathcal{S} \vdash \rho\langle y, \mathcal{I}nit(t_2)\rangle.\phi\langle x, e_2\rangle.m]\ \text{[e-@-pam]}$$

Cette fonction $\mathcal{I}nit$ prend un type fermé quelconque et retourne un terme correctement typé. Elle est définie par les deux tableaux suivants

t	**None**	**num**	**string**	**bool**	t_1 **in** $\{e_1 \cdots e_n\}$
$\mathcal{I}nit(t)$	**none**	0	"	**false**	e_1

t	$t_1 \to t_2$	$t_1 \Longrightarrow t_2$
$\mathcal{I}nit(t)$	$\lambda x : t_1 . \mathcal{I}nit(t_2)$	$\nabla x : t_1, y : t_2 . \{\ell : y = \mathcal{I}nit(t_2)\}$

Cette fonction possède la propriété suivante :

Propriété 5.13 *correction de la fonction* $\mathcal{I}nit$

$$\textit{pour tout type } t \textit{ tel que } \gamma \ \rhd \ t \textit{ alors } \gamma \rhd \mathcal{I}nit(t) : t$$

Preuve : Par induction sur la structure de t □

De plus, nous devons redefinir l'équation [e-@-err3] afin de prendre en compte les appels de machines abstraites.

$$\frac{v_1 \not\equiv \lambda x : t . e \quad v_1 \not\equiv \nabla x : t_1, y : t_2 . e_1}{[\mathcal{S} \vdash v_1(v_2)] \to [\mathcal{S} \vdash \textbf{error}]} \ \text{[e-@-err3]}$$

5.7.4 Correction du système de types

Proposition 5.14 *Préservation du type d'un terme* $\rho\langle x, e_1\rangle . e_2$

$$\gamma \rhd \rho\langle x, e_1\rangle . e_2 : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash \rho\langle x, e_1\rangle . e_2] \to [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : similaire la démonstration de la propriété **??** pour les termes $\phi\langle x, e_1\rangle . e_2$. La différence correspond au paramètre de retour, transition [e-eval3] □

Proposition 5.15 *Préservation du type d'un terme* $\nabla x : t_1, y : t_1 . e$

$$\gamma \rhd \nabla x : t_1, y : t_1 . e : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash \nabla x : t_1, y : t_1 . e] \to [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : sans difficultés □

Proposition 5.16 *Préservation du type d'un terme* **var** $x : t = e . m$

$$\gamma \rhd \textbf{var } x : t = e . m : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash \textbf{var } x : t = e . m] \to [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : Sans difficulté, en utilisant la propriété **??** pour les termes $\phi\langle x, e_1\rangle . e_2$. □

La proposition suivante demande une démonstration plus "articulée".

Proposition 5.17 *Préservation du type d'un terme $\nabla x:t_1, y:t_2.m(e_2)$ (∇-application)*

$$\gamma \rhd \nabla x:t_1, y:t_2.m(e_2) : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S}:\gamma \\ [\mathcal{S} \vdash \nabla x:t_1, y:t_2.m(e_2)] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}':\gamma \end{array} \right.$$

Preuve : L'unique transition concernée [e-@-pam] laisse le contexte invariant. Nous travaillerons donc uniquement sur le typage. L'hypothèse que le terme est bien typé se développe de manière unique en

$$\dfrac{\dfrac{\dfrac{\gamma \rhd t_1 \quad \gamma \rhd t_2}{\gamma \rhd t_1 \Longrightarrow t_2}\,[a] \quad x \neq y \quad x,y \notin \text{dom}(\gamma) \quad \gamma, x:t_1, y:t_2 \rhd m : \textbf{Unit}}{\gamma \rhd \nabla x:t_1, y:t_2.m : t_1 \Longrightarrow t_2}\,[b] \quad \gamma \rhd e_2 : t_1}{\gamma \rhd \nabla x:t_1, y:t_2.m(e_2) : t_2}\,[c]$$

[a] [t-lam]
[b] [te-pam]
[c] [te-@-pam]

Or la transition [e-@-pam] produit un terme dont le typage est obtenu de manière unique par l'arbre suivant

$$\dfrac{y \notin \text{dom}(\gamma) \quad \gamma \rhd \mathcal{I}nit(t_2) : t_2 \quad \dfrac{x \notin \text{dom}(\gamma, y:t_2) \quad \gamma, y:t_2 \rhd e_2 : t_1 \quad \gamma, y:t_2, x:t_1 \rhd m : \textbf{Unit}}{\gamma, y:t_2 \rhd \phi\langle x, e_2\rangle.m : \textbf{Unit}}\,[a]}{\gamma \rhd \rho\langle y, \mathcal{I}nit(t_2)\rangle.\phi\langle x, e_2\rangle.m : t_2}\,[b]$$

[a] [te-context]
[b] [te-eval]

ce qui constitue notre preuve, car dans l'ordre des feuilles du numérateur

$$\begin{array}{rrcll} 1 & x,y \notin \text{dom}(\gamma) & \Rightarrow & y \notin \text{dom}(\gamma) & \\ 2 & \gamma \rhd t_2 & \Rightarrow & \gamma \rhd \mathcal{I}nit(t_2) : t_2 & [5.13] \\ 3 & x \neq y \wedge x,y \notin \text{dom}(\gamma) & \Rightarrow & x \notin \text{dom}(\gamma, y:t_2) & \\ 4 & \gamma \rhd e_2 : t_1 \wedge y \notin \text{dom}(\gamma) & \Rightarrow & \gamma, y:t_2 \rhd e_2 : t_1 & \\ 5 & \gamma, x:t_1, y:t_2 \rhd m : \textbf{Unit} & \Rightarrow & \gamma, y:t_2, x:t_1 \rhd m : \textbf{Unit} & \end{array}$$

□

5.8 Synthèse

Nous avons proposé des extensions graduelles au langage *Circus* visant à enrichir les primitives de contrôle (que l'on oppose généralement aux données), tout en conservant les propriétés essentielles du noyau, notamment la correction du système de types.

Dans la première étape, de nouvelles instructions impératives associées à des déclarations de variables locales ont permis d'étendre de modèle calculatoire du fonctionnel à l'impératif. De plus une (unique) primitive d'itération permet de dépasser les limitations initiales dues à l'absence de fonctions récursives. D'autre part, l'utilisation des particularités des types énumérés autorise une detection des itérations pathologiques au niveau même du contrôle de types.

La seconde étape s'est proposée de poursuivre l'enrichissement de l'expressivité du langage, en intégrant des règles associées à des opérateurs de filtrage, l'ensemble ouvrant la voie vers des approches algorithmiques apparentées à la réécriture, dont l'interêt est bien connu dans le domaine du traitement des langages et de la transformation de structures en général. Ici encore, notre apport a été d'intégrer un modèle déclaratif au noyau fonctionnel et impératif, tout en garantissant la cohérence du système de types et de la sémantique opérationnelle.

D'une manière assez élégante, les opérateurs de filtrage ont pu être utilisés pour proposer dans la troisième partie, une extension visant à intégrer un modèle de coordination de type *Linda*. La présence d'une mémoire de coordination commune à tous les processus communicants à réclamé une modification profonde du système de transition, et une extension plus légère de la syntaxe et du système de types. Toutefois, cette modification n'a pas remis en cause les résultats antérieurs liés à la sémantique opérationnelle. En effet, la mémoire $\mathcal{M}$ peut se voir comme une constante dans les transitions sans rapport avec la coordination. Muni de quatre primitives synchrone ou asynchrone pour consulter la mémoire de manière associative, le langage offre des moyens de synchronisation et de coordination aptes à modèliser les complexes échanges d'information que nous avions mis en évidence dans les applications visuelles fortements interactives.

L'étape suivante, logiquement, consistait à introduire de la concurrence. Un simple opérateur permet de lancer de manière synchrone deux activités, dont la terminaison n'est pas synchronisée. Si l'expressivité résultante n'est pas maximale (pas de possibilité de lancer un nombre variable de processus en parallèle, par exemple), elle nous a semblé suffisante pour la classe d'application visée. L'expérimentation nous permettra de juger de la nécessité de proposer par exemple, un opérateur similaire au *eval* de *Linda*, qui lance de manière asynchrone un processus ecrivant son résultat dans la mémoire de coordination. L'introduction du parallèlisme dans le système de transition semble facile. Pourtant, l'ensemble du système de transition a été soigneusement conçu pour cela depuis les premières définitions du noyau fonctionnel afin de définir avec précision l'atomicité des transitions et la finesse de l'entrelacement. Ainsi, en l'absence de parallèlisme, la sémantique du test peut se définir par

$$\frac{[\mathcal{S} \vdash e_1] \twoheadrightarrow\!\circ\ \ [\mathcal{S}' \vdash \textbf{true}]}{[\mathcal{S} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3] \rightarrow [\mathcal{S}' \vdash e_2]} \qquad \frac{[\mathcal{S} \vdash e_1] \twoheadrightarrow\!\circ\ \ [\mathcal{S}' \vdash \textbf{false}]}{[\mathcal{S} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3] \rightarrow [\mathcal{S}' \vdash e_3]}$$

Dans un environnement concurrent, cette définition ne modèlise pas l'entrelacement temporel, puisque que l'évaluation de la condition booléene, qui requiert un -potentiellement- grand nombre d'étapes, est vu comme une seule transition atomique. Il faut donc introduire une règle qui modèlise les transitions atomiques permettant d'évaluer pas à pas la condition booléene, et deux règles qui s'appliquent quand le

booléen est normalisé :

$$\frac{[\mathcal{S} \vdash e_1] \rightarrow [\mathcal{S}' \vdash e_1']}{[\mathcal{S} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S}' \vdash \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3]}$$

$$[\mathcal{S} \vdash \textbf{if true then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S} \vdash e_2]$$

$$[\mathcal{S} \vdash \textbf{if false then } e_2 \textbf{ else } e_3] \rightarrow [\mathcal{S} \vdash e_3]$$

L'extension de l'expressivité du langage s'est ensuite poursuivi avec ce que nous avons appelé les systèmes d'actions, dont la vocation est de regrouper des règles (ou autres actions impératives) en une collection ordonnée. Intuitivement, ces systèmes définissent des actions qui sont réalisées l'une après l'autre, dans l'ordre de leur déclaration, et tant que l'action est évaluée à **none**. Bien sur, cette construction modèlise parfaitement les systèmes de réécriture séquentiels (dans ce cas, les actions sont toutes des règles) où l'ordre d'application est déterministe. De même, la réécriture conditionnelle est directement modèlisable. Le lecteur se convaincra de la grande expressivité proposée en remarquant que la partie droite d'une règle est composée de toute action impérative simple ou composite (acions séquentielles ou concurrentes,itérations,tests,déclarations,...) mais également d'autres règles ou systèmes d'actions qui seront évalués dans le contexte défini par l'évaluation de la règle principale. Il est donc possible de structurer la réécriture en décomposant les étapes de filtrage.

Les systèmes d'action sont utilisés dans la toute dernière extension proposée, la plus importante de toutes, que nous avons nommé *machines abstraites polymorphes* et qui constituent le cœur de notre contribution. Leur première fonction est de fournir une abstraction similaire à l'abstraction λ, que nous avons noté ∇, mais dédiée aux instructions impératives et déclaratives. Notons d"ailleur que la λ-application et la ∇-application possèdent la même forme syntaxique : $e_1(e_2)$. Le nom *machine* fait référence au fait que les abstractions ∇ modèlisent des machines à états finis (pas de récursion), et *abstraite* au fait que ces machines n'ont évidemment aucune réalité physique. La justification de *polymorphe* ne viendra qu'avec les developpements du chapitre sur la composition, ou il sera montré que nos abstraction ∇ exhibent un comportement polymorphe lors de leur composition, en ce sens qu'un code initialement écrit pour une signature donnée (un type) peut encore être utilisé lorsque la signature évolue. Indépendament de ce polymorphisme "compositionnel", nous bénéficions du polymorphisme naturellement induit par la relation de sous-typage, qui permet d'utiliser l'application avec tous les arguments qui possèdent un type compatible. Ce que nous pouvons retenir ici sont les contraintes sur la forme syntaxique, contraintes imposées par la grammaire formelle : principalement, le corps d'une *map* est toujours constitué d'un système d'actions, ceci dans le but de garantir les propriétés compositionelles dont nous aurons besoin ultérieurement.

Types de données structurés

6.1 Introduction

Ce chapitre vise à étendre l'expressivité du langage *Circus*, non pas au niveau du contrôle, comme dans le chapitre précédent, mais au niveau des données. Les types de données que nous proposons ont été choisis sur une base expérimentale, en ce sens que nous avons extrait de nos applications prototypes une palette suffisamment variée pour pouvoir modéliser la pluspart des structures de données nécessaires dans le domaine du traitement des langages. Aucun d'eux n'est innovant en soi, mais leur intégration au système de types, les primitives de filtrage qui leur sont associées et le réalisme qu'ils apportent par leur synthèse à un langage originellement "formel" constitueront les principales contributions de cette partie de notre étude.

6.2 Préliminaires

6.2.1 Types structurés et énumérations

Avec l'introduction de types de données structurés, beaucoup plus riches que les types de base, il est nécessaire de raffiner la règle de typage [te-enum], qui concerne les types énumérés, et que nous rappelons ici :

$$\frac{\gamma \rhd t \textbf{ in } \{e_1, \cdots, e_n\} \quad \gamma \rhd e : t \quad \exists i \in [1, n] \,.\, \gamma \rhd e \sim e_i}{\gamma \rhd e : t \textbf{ in } \{e_1, \cdots, e_n\}} \text{ [te-enum]}$$

Jusqu'à présent, l'existence d'une valeur normale commune entre deux termes confluents (ici e et e_i) suffisait pour affirmer qu'un terme e appartenait bien au domaine de valuation. Si on considère par exemple un ensemble $\{0, 1\}$, par essence non ordonné, nous ne pouvons pas écrire :

$$\gamma \rhd \{0, 1\} \sim \{1, 0\}$$

car, à l'évidence, ces deux formes normales sont distinctes. Pourtant, sémantiquement, il y a bien égalité, ce que confirmerait l'évaluation du terme $\{0, 1\} == \{1, 0\}$ qui fournirait comme résultat **true** d'après la sémantique opérationnelle que nous verrons plus loin. La conséquence est que nous ne pouvons pas affirmer :

$$\gamma \rhd \{0, 1\} : \top \textbf{ in } \{\{1, 0\}\}$$

ce qui est très restrictif du point de vue programmatique. Nous proposons de redéfinir légèrement [te-enum] pour dépasser cette limitation

$$\frac{\gamma \rhd t \textbf{ in } \{e_1, \cdots, e_n\} \quad \gamma \rhd e : t \quad \exists i \in [1, n] \,.\, \gamma \rhd e \approx e_i}{\gamma \rhd e : t \textbf{ in } \{e_1, \cdots, e_n\}} \text{ [te-enum]}$$

$$\frac{\gamma \rhd e_1 \rightsquigarrow v_1 \quad \gamma \rhd e_2 \rightsquigarrow v_2 \quad [\emptyset \vdash v_1 == v_2] \rightarrow [\emptyset \vdash \textbf{true}]}{\gamma \rhd e_1 \approx e_2} \text{ [te-equal]}$$

Notons que notre nouvelle règle n'est qu'une simple généralisation de la règle d'origine. D'autre part, comme v_1 et v_2 sont des termes normaux, le terme $v_1 == v_2$ est confluent (cf sémantique opérationnelle, [e-eq1] et [e-eq2]). Le terme $v_1 == v_2$ est calculable sur tous les termes du langage, car l'opérateur $==$ est défini (polymorphisme d'opérateur). De plus il ne produit jamais d'erreur. Enfin, notons que ce qui est vrai dans un contexte d'exécution vide l'est dans tout contexte $\mathcal{S}$.

6.2.2 Union de types

Nous avons pu apprécier l'importance de cet opérateur. L'introduction de types structurés amène une subtilité nouvelle dans le calcul du plus petit super-type commun. Considérons le type ensemble, noté $\{t\}$. Un type énuméré $t_1 \equiv \top$ **in** $\{\{0,1\},\{1\}\}$ est bien formé. Si l'on consière le type ensemble suivant $t_2 \equiv \{$**num in** $\{0,1\}\}$, la définition "actuelle" de $t_1 \oplus t_2$ nous permettrait de considérer $\{$**num in** $\{0,1\}\}$ comme le plus petit super-type commun, car t_1 est un sous-type de t_2. Si à présent nous considérons $t_3 \equiv \top$ **in** $\{\{0,1\},\{2\}\}$, qui n'est plus un sous-type de t_2, nous avons $t_3 \oplus t_2 = \top$, en suivant la définition adoptée pour les autres types de base. Or, ce résultat n'est pas minimal, en ce sens qu'il existe $t \equiv \{$**num in** $\{0,1,2\}\}$, super-type commun, mais tel que $\emptyset \rhd t \preccurlyeq \top$. Ainsi, la définition de $\oplus$, mais aussi de l'algorithme associé devra se complexifier pour prendre en compte cette particularité.

6.2.3 Types principaux

Nous avons introduit dans le chapitre précédent la notion de type minimal d'un terme, qui vient naturellement enrichir la notion de terme bien typé. Avec le filtrage de termes possédant des types structurés plus complexes que les types de base, nous allons avoir besoin d'un nouvel outil pour raisonner sur les types, qui permette en un certain sens de nous ramener à l'essentiel de l'information de type, en faisant abstraction de la notion de sous-typage par énumération. Nous proposons donc d'assigner des *types principaux* aux termes bien typés, relation logique représentée par la notation suivante ("le terme e a pour type principal t")

$$\gamma \rhd e \preccurlyeq:: t$$

Au moyen des équations suivantes

$$t \in \{\bot, \textbf{None}, \textbf{num}, \textbf{bool}, \textbf{string}, \textbf{Unit}\} \quad \frac{\gamma \rhd e : t}{\gamma \rhd e \preccurlyeq:: t} \text{ [te-p]}$$

$$\frac{\gamma \rhd e \preccurlyeq: \top}{\gamma \rhd e \preccurlyeq:: \top} \text{ [te-p2]}$$

$$\frac{\gamma, x : t_1 \rhd x \preccurlyeq:: t \quad \gamma, x : t_1 \rhd e \preccurlyeq:: t'}{\gamma \rhd \lambda x : t_1 . e \preccurlyeq:: t \to t'} \text{ [te-p3]}$$

$$\frac{\gamma, x : t_1 \rhd x \preccurlyeq:: t \quad \gamma, x : t_1, y : t_2 \rhd y \preccurlyeq:: t' \quad \gamma, x : t_1, y : t_2 \rhd m : \textbf{Unit}}{\gamma \rhd \nabla x : t_1, y : t_2 . m \preccurlyeq:: t \Longrightarrow t'} \text{ [te-p4]}$$

Le lecteur peut s'assurer que ce jeu de règles est complet par rapport à la syntaxe des types et des termes. D'autre part, il montre que seul un terme bien typé peut avoir un type principal.

Enfin, nous pouvons généraliser le typage des opérations de filtrage avec une nouvelle règle qui précisément utilise (et justifie) le typage principal

$$\frac{\gamma \rhd f \preccurlyeq:: t \quad \gamma \rhd e : t}{\gamma \rhd e \sharp f \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}} \text{ [te-match]}$$

Notons que cette règle combinée aux règles qui seront définies dans ce chapitre s'applique au filtrage des types de base, et donc peuvent être considérées comme leur généralisation (cf [te-cat], [te-match-free],[te-match-free2] et [te-invar] [1]).

6.2.4 Calcul du type principal - calcul du type d'une opération de filtrage

Le premier algorithme est une fonction qui prend un terme ou un filtre quelconque et calcule en retour son type principal, dans tout contexte de typage bien formé γ. Si le terme ou le filtre n'est pas bien typé, la fonction retourne la valeur distinguée **error**.

Définition 6.1 *Algorithme de calcul du type principal d'un terme ou d'un filtre*

$\mathcal{F}ndP \quad : \quad \gamma \times e \to t \cup \{\textbf{error}\}$

$\mathcal{F}ndP(\gamma, e) \quad =$

if $(e \equiv x)$ *then return* $\mathcal{F}ndP(\gamma, \gamma(x))$
if $(e \equiv \lambda x\!:\!t.e_1)$ *then*
  $t_1 = \mathcal{F}ndP(\gamma + \{x : t\}, x)$
  $t_2 = \mathcal{F}ndP(\gamma + \{x : t\}, e_1)$
  if $t_1 \equiv$ **error** *or* $t_2 \equiv$ **error** *then return* **error**
  return $t_1 \to t_2$
if $(e \equiv \nabla x\!:\!t_1, y\!:\!t_2.m)$ *then*
  $t_1 = \mathcal{F}ndP(\gamma + \{x : t_1, y : t_2\}, x)$
  $t_2 = \mathcal{F}ndP(\gamma + \{x : t_1, y : t_2\}, y)$
  if $t_1 \equiv$ **error** *or* $t_2 \equiv$ **error** *then return* **error**
  if $\mathcal{C}hk(\gamma + \{x : t_1, y : t_2\}, m, \textbf{Unit}) = \textbf{false}$ *then return* **error**
  return $t_1 \Longrightarrow t_2$
for $t \in \{\bot, \textbf{None}, \textbf{num}, \textbf{string}, \textbf{bool}, \textbf{Unit}\}$ **do**
  if $\mathcal{C}hk(\gamma, e, t)$ *then return* t
done
if $\mathcal{F}nd(\gamma, e) \equiv \top$ *then return* $\top$
return **error**

Notons que cet algorithme utilise $\mathcal{C}hk$ et $\mathcal{F}nd$, mais pas $\mathcal{S}ub$, ni $\mathcal{T}ype$ de manière directe.

Le calcul du type de $e\sharp f$ est réalisé via une implémentation de la règle générique [te-match]. Cette extension de $\mathcal{F}nd$ utilise $\mathcal{F}ndP$:

Définition 6.2 *Algorithme de calcul du type pour un terme* $e\sharp f$ *(extension de 4.69)*

if $e \equiv e\sharp f$ *then*
  $t = \mathcal{F}ndP(\gamma, f)$
  if $t \equiv$ **error** *then return* **error**
  if $\mathcal{C}hk(\gamma, e, t) \equiv$ **true** *then return* **bool in** $\{\textbf{false}, \textbf{true}\}$
  return **error**

Par des arguments et techniques similaires à ceux déjà utilisés auparavant, on peut montrer que l'algorithme est correct et termine dans tous les cas.

[1] Notons que dans [te-invar], la relation $\gamma \rhd t_1 \approx t_2$ était une "approximation" de la notion de type principal

6.3 Structures

Le type de données que nous présentons dans cette section correspond à une extension de la proposition de L. Cardelli et P. Wegner [24]. Il permet de composer des types de données hétérogènes, tout en identifiant les membres par des labels uniques. Il s'apparente au type *record* de *Pascal* ou *struct* du langage *c*, tout en étant adapté au système de type, et plus précisemment, à la relation de sous-typage et aux opérations de filtrage. Par ailleurs, l'ordre des membres n'est pas significatif dans les opérations de comparaison, d'affectation et de filtrage, ce qui confère une plus grande souplesse dans la manipulation des termes de ce type. L'ordre est uniquement significatif dans l'évaluation des termes (déterministe).

6.3.1 Syntaxe

e	::=	$\langle \ell_1 = e_1, \cdots, \ell_n = e_n \rangle$	*Structure avec membres étiquetés*
e	::=	$e.\ell$	*Lecture d'un membre*
e	::=	$x.\ell = e$	*Modification d'un membre*
t	::=	$\langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle$	*Type structuré avec membres étiquetés*

6.3.2 Règles de typage

L'égalité de deux structures est typée via la règle générique [te-eq]. Pour les autres opérations les définitions sont les suivantes :

$$\ell_1, \cdots, \ell_n \text{ uniques} \quad \frac{\gamma \rhd t_1 \quad \cdots \quad \gamma \rhd t_n}{\gamma \rhd \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle} \text{ [t-struct]}$$

$$\frac{\forall i \in [1 \cdots n], \quad \exists j \in [1 \cdots n+k] \mid \ell'_i = \ell_j \wedge \gamma \rhd t_j \preccurlyeq t'_i}{\gamma \rhd \langle \ell_1 : t_1, \cdots, \ell_{n+k} : t_{n+k} \rangle \preccurlyeq \langle \ell'_1 : t'_1, \cdots, \ell'_n : t'_n \rangle} \text{ [st-struct]}$$

$$\frac{\gamma \rhd e_1 : t_1 \quad \cdots \quad \gamma \rhd e_n : t_n}{\gamma \rhd \langle \ell_1 = e_1, \cdots, \ell_n = e_n \rangle : \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle} \text{ [te-struct]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: t_1 \quad \cdots \quad \gamma \rhd e_n \preccurlyeq:: t_n}{\gamma \rhd \langle \ell_1 = e_1, \cdots, \ell_n = e_n \rangle \preccurlyeq:: \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle} \text{ [te-p-struct]}$$

$$\frac{\gamma \rhd e : \langle \ell : t \rangle}{\gamma \rhd e.\ell : t} \text{ [te-memb]}$$

$$\frac{\gamma \rhd x : \langle \ell : t \rangle \quad \gamma \rhd e : t}{\gamma \rhd x.\ell = e : \textbf{Unit in } \{\textbf{unit}\}} \text{ [te-memb-w]}$$

$$t_1, \cdots, t_n \neq \top \quad \frac{\gamma \rhd e_1 : \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle \quad \gamma \rhd e_2 : \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle}{\gamma \rhd e_1 \leq e_2 : \textbf{bool}} \text{ [te-leq-struct]}$$

Notons encore que l'ordre de déclaration des étiquettes n'a pas d'importance, tant pour la relation de sous-typage, que pour le typage des termes.

6.3.3 Union minimale et intersection maximale

L'union minimale de deux types "structure" est une structure dont 1/ les étiquettes sont communes aux deux opérandes, 2/ les membres associés sont l'union minimale des membres respectifs. Lorsqu'il n'y a pas d'étiquettes en commun, le résultat est le type $\top$.

Dans le cas où l'un des type est une structure (t_1), et l'autre une énumération (t_2), nous retrouvons potentiellement le problème évoqué dans le préliminaire, si t_2 n'est pas sous-type de t_1. La définition propose alors une solution générale, dont le schéma consiste à traiter chaque terme énuméré par t_2 de façonà déterminer un type "presque" minimal, non énuméré. Ainsi un terme $\langle \ell = 10, m =' s' \rangle$ se voit affecter le type $\langle \ell : \textbf{num in } \{10\}, m : \textbf{string in } \{'s'\} \rangle$. Tous les types similaires pour tous les termes de l'énumération sont alors mis en union maximale avec t_1.

Les cas standards sont traités de manière identique aux autre types (cf tableau de définition de $\oplus$).

Définition 6.3 *Union minimale (extension aux types structure)*

Si $t_1 \equiv \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle$

Si $t_2 \equiv \langle \ell_1 : t'_1, \cdots, \ell_n : t'_n, \cdots, \ell'_k : t'_k \rangle$

$t_1 \oplus t_2 \stackrel{def}{=} \langle \ell_1 : t_1 \oplus t'_1, \cdots, \ell_n : t_n \oplus t'_n \rangle$

Sinon si $t_2 \equiv \langle \ell'_1 : t'_1, \cdots, \ell'_k : t'_k \rangle$ (*pas de membres en commun*)

$t_1 \oplus t_2 \stackrel{def}{=} \top$

Sinon si $t_2 \equiv t'_1$ **in** $\{e_1, \cdots, e_j\}$

- *si* $\gamma \rhd t_2 \preccurlyeq t_1$ *alors* $t_1 \oplus t_2 \stackrel{def}{=} t_1$
- *sinon si* $\forall i \in [1, j]$, $\begin{cases} e_i \equiv \langle m_{i1} = e_{i1}, \ldots, m_{ik} : e_{ik} \rangle \\ \gamma \rhd e_{i1} \preccurlyeq : t_{i1} \quad \cdots \quad \gamma \rhd e_{ik} \preccurlyeq : t_{ik} \end{cases}$

  $t_1 \oplus t_2 \stackrel{def}{=} (t_1 \oplus \langle m_{11} : t_{11}, \ldots \rangle \oplus \cdots \oplus \langle m_{j1} : t_{j1}, \ldots \rangle)$
- *sinon* $t_1 \oplus t_2 \stackrel{def}{=} \top$

Sinon si $t_1 \equiv t'_1$ **in** S' *et* $t_2 \not\equiv \langle \cdots \rangle$

$t_1 \oplus t_2 \stackrel{def}{=} t_2 \oplus t_1$ (*ce cas est traité précédement*)

Sinon si $t_2 \not\equiv t'_1$ **in** S' *et* $t_2 \not\equiv \langle \cdots \rangle$ *alors*

$t_1 \oplus t_2 \stackrel{def}{=} \top$

L'intersection maximale de deux structures correspond à une structure dont les étiquettes sont la réunion des étiquettes de chacune des opérandes. Lorsque des étiquettes sont communes le type associé est l'intersection maximale.

Définition 6.4 *Intersection maximale (extension aux types structure)*

$$\begin{array}{l}
\textit{Si } t_1 \equiv \langle \ell_1 : t_1, \ldots, \ell_n : t_n, \ldots \ell_{n+m} : t_{n+m} \rangle \\
\quad \textit{si } t_2 \equiv \langle \ell_1 : t'_1, \ldots, \ell_n : t'_n, \ldots, \ell'_{n+k} : t'_{n+k} \rangle \\
\qquad t_1 \otimes t_2 \stackrel{\textit{def}}{=} \\
\qquad \langle \ell_1 : t_1 \otimes t'_1, \ldots, \ell_n : t_n \otimes t'_n, \ell_{n+1} : t_{n+1}, \ldots \ell_{n+m} : t_{n+m}, \ell'_{n+1} : t'_{n+1}, \ldots, \ell'_{n+k} : t'_{n+k} \rangle \\
\\
\quad \textit{si } t_2 \equiv t \textbf{ in } \{e_1 \cdots e_n\} \\
\qquad \textit{si } S = \{e \in \{e_1, \ldots, e_n\} | \gamma \rhd e : t_1\} \neq \emptyset \quad t_1 \otimes t_2 \stackrel{\textit{def}}{=} t \textbf{ in } S \\
\qquad \textit{sinon } t_1 \otimes t_2 \stackrel{\textit{def}}{=} \bot \\
\\
\quad \textit{sinon } t_1 \otimes t_2 \stackrel{\textit{def}}{=} \bot
\end{array}$$

Proposition 6.5 *Preservation des propriétés de $\oplus$ et $\otimes$.*

Le calcul de l'union minimale et de l'intersection maximale de deux types quelconques étendu avec la définition 6.3 et 6.4 possède l'ensemble des propriétés établies dans le noyau.

Preuve : en annexe □

6.3.4 filtrage de structures

La syntaxe des filtres est étendue de manière à prendre en compte le nouveau type de donnée :

$$f \quad ::= \quad \langle \ell_1 = f_1, \cdots, \ell_n = f_n \rangle \quad \textit{filtre pour termes structurés}$$

L'équation générique [te-match] s'applique aux structures : notons qu'il est possible de filtrer un terme possédant plus de champs que le filtre lui-même (c'est un effet de la relation de sous-typage des structures, cf [st-struct]). Ainsi, par exemple, le jugement suivant peut être prouvé

$$x : \textbf{string} \rhd \langle \ell_1 = 10, \ell_2 =' s' \rangle \sharp \langle \ell_1 = ?x \rangle \preccurlyeq : \textbf{bool in } \{\textbf{true}, \textbf{false}\}$$

Toutefois, le système d'inférence n'est pas complet, car nous n'avons aucune règle pour déterminer le type principal d'un filtre structure, qui, du reste n'est pas un terme du langage (mais un "sous-terme", le terme étant $e \sharp \langle \ell = f, \cdots \rangle$). Nous proposons donc l'extension suivante. qui sera réutilisée par la suite :

$$\frac{\gamma \rhd f_1 \preccurlyeq :: t_1 \quad \cdots \quad \gamma \rhd f_n \preccurlyeq :: t_n}{\gamma \rhd \langle \ell_1 = f_1, \cdots, \ell_n = f_n \rangle \preccurlyeq :: \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle} \text{ [te-p-fstruct]}$$

Notre système d'inférence reste incomplet vis à vis du typage principal des filtres. Les équations suivantes permettent d'établir le type principal de tout filtre, soit directement (par exemple [te-p-free]) soit indirectement (comme [te-min-free2] combinée avec [te-p2]).

$$\frac{\gamma \rhd x \preccurlyeq : t}{\gamma \rhd ?x \preccurlyeq :: t} \text{ [te-p-free]} \quad \gamma \rhd ? \preccurlyeq : \top \text{ [te-min-free2]} \quad \frac{\gamma \rhd e \preccurlyeq :: t}{\gamma \rhd \%e \preccurlyeq :: t} \text{ [te-p-anti]}$$

Le groupe d'équations suivant prend en compte la concaténation de filtres, dont la seule utilisation à ce point de notre développement concerne le filtrage des chaînes de caractères. Il sera étendu avec les types séquence, ensemble et dictionnaire.

$$\frac{\gamma \rhd f_1 \preccurlyeq:: \textbf{string} \quad \gamma \rhd f_2 \preccurlyeq:: \textbf{string}}{\gamma \rhd f_1 \boxplus f_2 \preccurlyeq:: \textbf{string}}\ [\text{te-p-cat}] \qquad \frac{\gamma \rhd f_1 \preccurlyeq:: \textbf{string} \quad \gamma \rhd f_2 \preccurlyeq:: \top}{\gamma \rhd f_1 \boxplus f_2 \preccurlyeq:: \textbf{string}}\ [\text{te-p-cat1}]$$

$$\frac{\gamma \rhd f_1 \preccurlyeq:: \top \quad \gamma \rhd f_2 \preccurlyeq:: \textbf{string}}{\gamma \rhd f_1 \boxplus f_2 \preccurlyeq:: \textbf{string}}\ [\text{te-p-cat2}]$$

On souhaite par exemple qu'un filtre $\%10$ puisse être appliqué à une variable x telle que $\gamma \rhd x \preccurlyeq:$ **num**. Le problème de typage revient à prouver le jugement suivant :

$$x : \textbf{num} \rhd \langle \ell = x \rangle \sharp \langle \ell = \%10 \rangle \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}$$

ce qui se fait par (on suppose que γ contient $x :$ **num**)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\gamma \rhd 10 : \textbf{num}}\,[a]}{\gamma \rhd 10 \preccurlyeq:: \textbf{num}}\,[b]}{\gamma \rhd \%10 \preccurlyeq:: \textbf{num}}\,[c]}{\gamma \rhd \langle \ell = \%10 \rangle \preccurlyeq:: \langle \ell : \textbf{num} \rangle}\,[d] \quad \dfrac{\dfrac{}{\gamma \rhd x : \textbf{num}}\,[e]}{\gamma \rhd \langle \ell = x \rangle : \langle \ell : \textbf{num} \rangle}\,[f]}{\gamma \rhd \langle \ell = x \rangle \sharp \langle \ell = \%10 \rangle \preccurlyeq: \textbf{bool in } \{\textbf{true}, \textbf{false}\}}\,[g]$$

[a] [te-num]
[b] [te-p]
[c] [te-p-anti]
[d] [te-p-fstruct]
[e] [te-env]
[f] [te-struct]
[g] [te-match]

6.3.5 Sémantique

La règle [e-struct-norm] dit qu'un terme structure est normal lorsque tous ses membres sont normaux. La règle [e-struct] exprime que la réduction du terme se fait dans l'ordre d'occurrence des sous-termes (si l'ordre n'est pas signifiant pour la relation de sous-typage et dans les opérations d'affectation et de comparaison, il a une importance pour déterminer sans ambiguité comment un terme structure se réduit).

$$\frac{[\mathcal{S} \vdash e_i] \to [\mathcal{S} \vdash e_i']}{[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_i = e_i, \cdots, \ell_n = e_n \rangle] \to [\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_i = e_i', \cdots, \ell_n = e_n \rangle]}\ [\text{e-struct}]$$

$$[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle] \to [\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle]\ [\text{e-struct-norm}]$$

$$[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_i = \textbf{error}, \cdots, \ell_n = e_n \rangle] \to [\mathcal{S} \vdash \textbf{error}]\ [\text{e-struct-err}]$$

L'accés aux membres est traité sans surprise avec une phase de reduction du terme principal, puis une phase de lecture proprement dite.

$$\frac{[\mathcal{S} \vdash e] \to [\mathcal{S} \vdash e']}{[\mathcal{S} \vdash e.\ell] \to [\mathcal{S} \vdash e'.\ell]} \text{ [e-memb1]}$$

$$[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell = v, \cdots, \ell_n = v_n \rangle.\ell] \to [\mathcal{S} \vdash v] \text{ [e-memb2]}$$

$$\ell \notin \{\ell_1, \cdots, \ell_n\} \quad [\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle.\ell] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-memb-err]}$$

$$[\mathcal{S} \vdash \textbf{error}.\ell] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-memb-err2]}$$

La modification d'un membre ne peut se faire que via une référence. Ce premier est mis à jour en mémoire après réduction de la nouvelle valeur.

$$\frac{[\mathcal{S} \vdash e] \to [\mathcal{S} \vdash e']}{[\mathcal{S} \vdash x.\ell = e] \to [\mathcal{S} \vdash x.\ell = e']} \text{ [e-memb-w]}$$

$$[\mathcal{S}, x = \langle \ldots, \ell = v', \ldots \rangle \vdash x.\ell = v] \to [\mathcal{S}, x = \langle \ldots, \ell = v, \ldots \rangle \vdash \textbf{unit}] \text{ [e-memb-w2]}$$

$$\ell \notin \{\ell_1, \cdots, \ell_n\}$$
$$[\mathcal{S}, x = \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle \vdash x.\ell = v] \to [\mathcal{S}, x = \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle \vdash \textbf{error}] \text{ [e-memb-w-err1]}$$

$$\neg(\emptyset \rhd v' : \langle \ell : u' \rangle)$$
$$[\mathcal{S}, x = v' \vdash x.\ell = v] \to [\mathcal{S}, x = v' \vdash \textbf{error}] \text{ [e-memb-w-err2]}$$

$$[\mathcal{S} \vdash x.\ell = \textbf{error}] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-memb-w-err3]}$$

Nous définissons également l'égalité et l'inégalité de deux structures normalisées par les deux groupes d'équations suivants (notons que l'ordre des étiquettes n'a pas d'incidence)

$$\frac{\forall i, j \in [1, n], \quad \ell_i = \ell'_j \quad \Rightarrow \quad [\mathcal{S} \vdash v_i == v'_j] \to [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle == \langle \ell'_1 = v'_1, \cdots, \ell'_n = v'_n \rangle] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-eq-struct]}$$

$$\frac{\exists i, j \in [1, n] \text{ tels que } \ell_i = \ell'_j \quad \wedge \quad [\mathcal{S} \vdash v_i == v'_i] \to [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle == \langle \ell'_1 = v'_1, \cdots, \ell'_n = v'_n \rangle] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-struct2]}$$

$$\frac{k \neq n}{[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle == \langle \ell'_1 = v'_1, \cdots, \ell'_k = v'_k \rangle] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-struct3]}$$

$$\frac{\forall i \in [1, n], \ \exists j \in [1, n] \text{ tel que } [\mathcal{S} \vdash v_i <= v'_j] \to [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle <= \langle \ell'_1 = v'_1, \cdots, \ell'_{n+k} = v'_{n+k} \rangle] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-ineq-struct]}$$

$$\frac{\exists i, j \in [1, n] \text{ tels que } \ell_i = \ell'_j \quad \wedge \quad [\mathcal{S} \vdash v_i <= v'_j] \to [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash \langle \ell_1 = v_1, \cdots, \ell_n = v_n \rangle <= \langle \ell'_1 = v'_1, \cdots, \ell'_{n+k} = v'_{n+k} \rangle] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-ineq-struct2]}$$

Enfin, la sémantique de l'opération de filtrage est décrite par [e-match-struct]. Notons la propagation des contextes dans le numérateur. La règle [e-fstruct] décrit simplement la réduction d'un filtre structure : similairement à [e-struct], dans l'ordre d'occurence des membres.

$$\frac{[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1}] \rightarrow [\mathcal{S}^{(1)} \vdash \textbf{true}] \quad \cdots \quad [\mathcal{S}^{(n-1)} \vdash v_n \sharp \boldsymbol{f_n}] \rightarrow [\mathcal{S}^{(n)} \vdash \textbf{true}]}{[\mathcal{S} \vdash \langle \ell_1 : v_1, \cdots, \ell_{n+k} : v_{n+k} \rangle \sharp \langle \ell_1 : \boldsymbol{f_1}, \cdots, \ell_n : \boldsymbol{f_n} \rangle] \rightarrow [\mathcal{S}^{(n)} \vdash \textbf{true}]} \text{ [e-match-struct]}$$

$$\frac{\begin{array}{c}\forall i \in [1,n] \\ [\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1}] \rightarrow [\mathcal{S}^{(1)} \vdash \textbf{true}] \quad \cdots \quad [\mathcal{S}^{(i-1)} \vdash v_i \sharp \boldsymbol{f_i}] \rightarrow [\mathcal{S}^{(i)} \vdash \textbf{false}]\end{array}}{[\mathcal{S} \vdash \langle \ell_1 : v_1, \cdots, \ell_{n+k} : v_{n+k} \rangle \sharp \langle \ell_1 : \boldsymbol{f_1}, \cdots, \ell_n : \boldsymbol{f_n} \rangle] \rightarrow [\mathcal{S}^{(i)} \vdash \textbf{false}]} \text{ [e-match-struct2]}$$

$$\frac{[\mathcal{S} \vdash f_i] \rightarrow [\mathcal{S}' \vdash f_i']}{[\mathcal{S} \vdash \langle \ell_1 = \boldsymbol{f}_1, \ldots, \ell_i = f_i, \ldots, \ell_n = f_n \rangle] \rightarrow [\mathcal{S}' \vdash \langle \ell_1 = \boldsymbol{f}_1, \ldots, \ell_i = f_i', \ldots, \ell_n = f_n \rangle]} \text{ [e-fstruct]}$$

6.3.6 Correction du système de type

Les termes de type "structure" respectent la propriété de préservation du type dans les réductions.

Proposition 6.6 *Préservation du type structure*

$$\begin{array}{l}\gamma \rhd \langle \ell_1 = e_1, \cdots, \ell_i = e_i, \cdots, \ell_n = e_n \rangle : t \\ \Rightarrow \\ \textit{pour tout } \mathcal{S}, \quad \textit{tel que} \quad \Vdash \mathcal{S} : \gamma \textit{ alors} \\ \left\{ \begin{array}{l} [\mathcal{S} \vdash \langle \ell_1 = e_1, \cdots, \ell_i = e_i, \cdots, \ell_n = e_n \rangle] \rightarrow [\mathcal{S}' \vdash \langle \ell_1 = e_1, \cdots, \ell_i = e_i', \cdots, \ell_n = e_n \rangle] \\ \gamma \rhd \langle \ell_1 = e_1, \cdots, \ell_i = e_i', \cdots, \ell_n = e_n \rangle : t \\ \Vdash \mathcal{S}' : \gamma \end{array} \right.\end{array}$$

Preuve :utilise la proposition 4.44 □

Proposition 6.7 *Préservation du type pour l'ensemble des opérations sur les structures.*

Les opérations d'accès à un membre , de modification d'un membre, de comparaison et de filtrage préservent la relation de typage comme défini en 4.44

Preuve :avec la proposition précédente, et les techniques utilisées dans le chapitre précédent □

Et enfin, nous devons étendre la fonction $\mathcal{I}nit$ qui calcule un terme par défaut pour tout type t bien formé (cette fonction est utilisée pour la ∇-application).

Définition 6.8 *Fonction de calcul d'un terme par défaut pour un type structure (extension de 5.7.3)*

$$\mathcal{I}nit(\langle \ell_1 : t_1, \ldots, \ell_n : t_n \rangle) \stackrel{def}{=} \langle \ell_1 : \mathcal{I}nit(t_1), \ldots, \ell_n : \mathcal{I}nit(t_n) \rangle$$

On peut montrer facilement que la propriété de correction $\gamma \quad \rhd \quad t \Rightarrow \gamma \rhd \mathcal{I}nit(t) : t$ (cf 5.13) est préservée.

6.3.7 Contrôle de type

Nous devons apporter des extensions aux algorithmes proposés dans le noyau du langage.

Définition 6.9 *Algorithme de test syntaxique de convergence (extension de 4.60)*

elif $(e \equiv \langle \ell_1 = e_1, \cdots, \ell_n = e_n \rangle)$ *then*
  for $e' \in e_1 \cdots e_n$ **do**
    if conv$(\gamma, e') ==$ **false** *then return* **false**
  done
  return **true**

Définition 6.10 *Algorithme de décision sur la validité des types (extension de 4.64)*

elif $(t \equiv \langle \ell_1 : t_1, \cdots, \ell_n : t_n \rangle)$ *then*
  for $t' \in t_1 \cdots t_n$ **do**
    if $\mathcal{T}ype(\gamma, t') ==$ **false** *then return* **false**
  done
  return **true**

Définition 6.11 *Algorithme de décision sur la relation de sous-typage (extension de 4.65)*

elif $(t_1 \equiv \langle \ell_1 : t_{11}, \cdots, \ell_n : t_{1n} \rangle)$ *then*
  if $(t_2 \equiv \langle \ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k} \rangle)$
    for $(\ell', t') \in \{\ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k}\}$ **do**
      if $\exists (\ell', t) \in \{\ell_1 : t_{11}, \cdots, \ell_n : t_{1n}\}$ *then*
        if $\mathcal{S}ub(\gamma, t, t') ==$ **false** *then*
          return **false**
        else
          return **false**
    done
    return **true**
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Définition 6.12 *Algorithme de contrôle de conformité (extension de $\mathcal{C}hk$ 4.6.7)*

if $(e \equiv \langle \ell_1 : e_1, \cdots, \ell_n : e_n \rangle)$ *then*
  if $(t \equiv \langle \ell'_1 : t_1, \cdots, \ell'_k : t_k \rangle)$
    for $(\ell', t') \in \{\ell'_1 : t_1, \cdots, \ell'_k : t_k\}$ **do**
      if $\exists (\ell', e') \in \{\ell_1 : e_1, \cdots, \ell_n : e_n\}$ *then*
        if $\mathcal{C}hk(\gamma, e', t') ==$ **false** *then*
          return **false**
        else
          return **false**
    done
    return **true**
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Dans l'algorithme suivant, l'énumération retournée en résultat pourrait être plus simple : $\top$ **in** $\{e\}$ conviendrait. Il faudrait alors s'assurer que le terme e est confluent et qu'il possède un type bien formé.

Définition 6.13 *Algorithme de calcul du type minimal d'un terme (extension de 4.69)*

if $(e \equiv \langle \ell_1 : e_1, \ldots, \ell_n : e_n \rangle)$ *then*
  $t_1 = \mathcal{F}nd(\gamma, e_1); \cdots; t_n = \mathcal{F}nd(\gamma, e_n)$
  if **error** $\in \{t_1, \cdots, t_n\}$ *then return* **error**
  return $\langle \ell_1 : t_1, \ldots, \ell_n : t_n \rangle$ **in** $\{e\}$

Dans l'algorithme suivant, nous avons incorporé le traitement de la concaténation de filtres pour les types séquence, ensemble et dictionnaire ($[t], \{t\}, \{t_1 : t_2\}$) qui seront developpés plus avant, par simplification.

Définition 6.14 *Algorithme de calcul du type principal d'un terme (extension de 6.1)*

if $(e \equiv ?)$ *then return* $\top$
if $(e \equiv ?x)$ *then return* $\mathcal{F}nd(\gamma, x)$
if $(e \equiv \%e_1)$ *then return* $\mathcal{F}nd(\gamma, e_1)$
if $(e \equiv f_1 \boxplus f_2)$ *then*
  $t_1 = \mathcal{F}ndP(\gamma, f_1)$
  $t_2 = \mathcal{F}ndP(\gamma, f_2)$
  if $t_1 \equiv$ **error** *or* $t_2 \equiv$ **error** *then return* **error**
  $t = \mathcal{P}rodType(\gamma, t_1, t_2)$
  if $t \equiv$ **string** *or* $t \equiv [t_1]$ *or* $t \equiv \{t_1\}$ *or* $t \equiv \{t_1 : t_2\}$ *then*
    return t
if $(e \equiv \langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle)$ *then*
  for $e_i \in \{e_1, \cdots, e_n\}$ **do**
    $t_i = \mathcal{F}ndP(\gamma, e_i)$
    if $t \equiv$ **error** *then return* **error**
  done
  return $\langle \ell_1 : t_1, \ldots, \ell_n : t_n \rangle$
if $(e \equiv \langle \ell_1 = f_1, \ldots, \ell_n = f_n \rangle)$ *then*
  for $f_i \in \{f_1, \cdots, f_n\}$ **do**
    $t_i = \mathcal{F}ndP(\gamma, f_i)$
    if $t \equiv$ **error** *then return* **error**
  done
  return $\langle \ell_1 : t_1, \ldots, \ell_n : t_n \rangle$

Dans la définition suivante, nous notons $\{\cdots, \ell_i : t_i, \cdots\}$ un ensemble de couples (ℓ_i, t_i), et $\langle s \rangle$ le type structure directement constructible à partir d'un tel ensemble s.

Définition 6.15 *Algorithme de calcul pour l'opérateur $\oplus$ (extension de 4.67)*

if $(t_1 \equiv \langle \ell_1 : t_{11}, \cdots, \ell_n : t_{1n} \rangle)$ *then*
  if $(t_2 \equiv \langle \ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k} \rangle)$
    $s = \{\ell_1, \cdots, \ell_n\} \cap \{\ell'_1, \cdots, \ell'_k\}$
    if $s == \emptyset$ *then*
      return $\top$
    $r = \emptyset$
    for $\ell \in s$ **do**
      if $(\ell, t) \in \{\ell_1 : t_{11}, \cdots, \ell_n : t_{1n}\}$ *and* $(\ell, t') \in \{\ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k}\}$ *then*
        $r = r \cup \{\ell : \mathcal{A}ddType(\gamma, t, t')\}$
    done
    return $\langle r \rangle$
  if $(t_2 \equiv t$ **in** $S)$ *then*
    if $\mathcal{S}ub(\gamma, t_2, t_1)$ *then return* t_1
    $t = \bot$
    for $e \in S$ **do**
      if $e \equiv \langle \ldots, m = ee, \ldots \rangle$ *then*
        $t = AddType(\gamma, t, \langle \ldots, m : \mathcal{F}nd(\gamma, ee), \ldots \rangle)$
      else $t = \top$
    done
    return $\mathcal{A}ddType(\gamma, t_1, t)$
  if $(t_2 \equiv \bot)$ *then*
    return t_1
  else return $\top$

L'extension de l'algorithme de calcul de l'intersection maximale est plus simple.

Définition 6.16 *Algorithme de calcul pour l'opérateur $\otimes$ (extension de 4.68)*

elif $(t_1 \equiv \langle \ell_1 : t_{11}, \cdots, \ell_n : t_{1n} \rangle)$ *then*
  if $(t_2 \equiv \langle \ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k} \rangle)$
    for $(\ell, t) \in \{\ell_1 : t_{11}, \cdots, \ell_n : t_{1n}\}$ **do**
      if $\exists t' | (\ell, t') \in \{\ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k}\}$ *then*
        $r = r \cup \{\ell : \mathcal{P}rodType(\gamma, t, t')\}$
      else $r = r \cup \{\ell : t\}$
    done
    for $(\ell, t) \in \{\ell'_1 : t_{21}, \cdots, \ell'_k : t_{2k}\}$ **do**
      if $\nexists (\ell, t') \in \{\ell_1 : t_{11}, \cdots, \ell_n : t_{1n}\}$ *then*
        $r = r \cup \{\ell : t\}$
    done
    return $\langle r \rangle$
  return $\bot$

6.4 Sequences

Le type “séquence” que nous proposons à présent modélise des collections ordonnées de valeurs ayant un type commun. Lorsque ce dernier est $\top$, cette collection peut comporter toute valeur bien typée (dans ce cas toutefois, les opérations permises sur les éléments seront plus restreintes). Les séquences vides sont licites. Les séquences peuvent être comparées ($==$ et $\leq$), concaténées et filtrées.

e	::=	$[e_1, \cdots, e_n]$	*Séquence ordonnée*
e	::=	$[\,]$	*Séquence vide*
t	::=	$[t_1]$	*Type séquence*

6.4.1 Règles de typage

Notons que l'égalité est traitée par la règle générique [te-eq], comme pour le type structure.

$$\frac{\gamma \rhd t}{\gamma \rhd [t]}\ \text{[t-seq]}$$

$$\frac{\gamma \rhd t_1 \preccurlyeq t_2}{\gamma \rhd [t_1] \preccurlyeq [t_2]}\ \text{[st-seq]}$$

$$\frac{\gamma \rhd e_1 : t \quad \cdots \quad \gamma \rhd e_n : t}{\gamma \rhd [e_1, \cdots, e_n] : [t]}\ \text{[te-seq]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: t_1 \quad \cdots \quad \gamma \rhd e_n \preccurlyeq:: t_n}{\gamma \rhd [e_1, \cdots, e_n] \preccurlyeq:: [t_1 \oplus \cdots \oplus t_n]}\ \text{[te-p-seq]}$$

$$\frac{}{\gamma \rhd [\,] : [\bot]}\ \text{[te-empty-seq]}$$

$$\frac{\gamma \rhd e_1 : [t] \quad \gamma \rhd e_2 : [t]}{\gamma \rhd e_1 + e_2 : [t]}\ \text{[te-plus-seq]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: [t_1] \quad \gamma \rhd e_2 \preccurlyeq:: [t_2]}{\gamma \rhd e_1 + e_2 \preccurlyeq:: [t_1 \oplus t_2]}\ \text{[te-p-plus-seq]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: [t] \quad \gamma \rhd e_2 \preccurlyeq:: [t]}{\gamma \rhd e_1 \leq e_2 : \textbf{bool}}\ \text{[te-leq-seq]}$$

6.4.2 Union minimale et intersection maximale

Nous retrouvons la même approche que pour les types structure, mais simplifiée toutefois, de par la nature du type séquence.

Définition 6.17 *Union minimale (extension aux types séquence)*

$$
\begin{array}{l}
\textit{if } t_1 \equiv [t_1] \\
\quad \textit{if } t_2 \equiv [t_2] \quad t_1 \oplus t_2 \stackrel{\textit{def}}{=} [t_1 \oplus t_2] \\
\quad \textit{if } t_2 \equiv t_1' \textbf{ in } \{e_1, \cdots, e_n\} \\
\quad \left\{ \begin{array}{l}
\textit{si } \gamma \rhd t_2 \preccurlyeq t_1 \textit{ alors } t_1 \oplus t_2 \stackrel{\textit{def}}{=} t_1 \\
\textit{sinon si } \forall i \in [1,n], \left\{ \begin{array}{l} e_i \equiv [e_{i1}, \ldots, e_{ik}] \\ \gamma \rhd e_{i1} \preccurlyeq\!: t_{i1} \quad \cdots \quad \gamma \rhd e_{ik} \preccurlyeq\!: t_{ik} \end{array} \right. \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} (t_1 \oplus [t_{11} \oplus \cdots t_{1k}] \oplus \cdots \oplus [t_{n1} \oplus \cdots t_{nk}]) \\
\textit{sinon } t_1 \oplus t_2 \stackrel{\textit{def}}{=} \top
\end{array} \right. \\
\textit{Sinon si } t_1 \equiv t_1' \textbf{ in } S' \textit{ et } t_2 \not\equiv [t] \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} t_2 \oplus t_1 \quad (\textit{ce cas est traité précédement}) \\
\\
\textit{Sinon si } t_2 \not\equiv t_1' \textbf{ in } S' \textit{ et } t_2 \not\equiv [t] \textit{ alors} \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} \top
\end{array}
$$

Définition 6.18 *Intersection maximale (extension aux types séquence)*

$$
\begin{array}{l}
\textit{Si } t_1 \equiv [t_{11}] \\
\quad \textit{si } t_2 \equiv [t_{21}] \quad t_1 \otimes t_2 \stackrel{\textit{def}}{=} [t_{11} \otimes t_{21}] \\
\quad \textit{si } t_2 \equiv t \textbf{ in } \{e_1 \cdots e_n\} \\
\qquad \textit{si } S = \{e \in \{e_1, \ldots, e_n\} | \gamma \rhd e : t_1\} \neq \emptyset \quad t_1 \otimes t_2 \stackrel{\textit{def}}{=} t \textbf{ in } S \\
\qquad \textit{sinon } t_1 \otimes t_2 \stackrel{\textit{def}}{=} \bot \\
\\
\quad \textit{sinon } t_1 \otimes t_2 \stackrel{\textit{def}}{=} \bot
\end{array}
$$

Proposition 6.19 *Préservation des propriétés de $\oplus$ et $\otimes$.*

Le calcul de l'union minimale et de l'intersection maximale de deux types quelconques étendu avec les définitions 6.17 et 6.18 possède l'ensemble des propriétés 'etablies dans le noyau.

Preuve : en annexe □

6.4.3 filtrage des séquences

Enfin, le filtrage de termes "séquence" s'applique sur des termes de même cardinal, en respectant l'ordre.

$f \quad ::= \quad [f_1, \cdots, f_n]$ *filtre pour termes "séquences"*

L' opérateur de concaténation $\boxplus$ est étendu pour prendre en compte les séquences de manière similaire aux chaînes de caractères. Ainsi, par exemple, le filtre $? \boxplus [?x, ?, ?y]$ permet de consulter l'élément $n-1$ et $n-3$ de toute séquence de cardinal ≥ 3 (on suppose que x et y on pour type minimal $\top$). D'un point de vue opérationnel, on peut écrire

$$[x = 0, y = 0 \vdash [1,2,3,4,5] \sharp ? \boxplus [?x, ?, ?y]] \twoheadrightarrow\!\circ \quad [x = 3, y = 5 \vdash \textbf{true}]$$

Les deux équations suivantes capturent toutes les situations de typage pour les filtres séquence et leur concaténation. Notons l'utilisation de $\otimes$ dans [te-seq2], et la généricité de [te-p-fcat] qui s'applique à tous types où $+$ possède une sémantique de concaténation : **string** et $[t]$, mais comme nous le verrons aussi pour les types ensembles et dictionnaires notés $\{t\}, \{t_1 : t_2\}$.

$$\frac{\gamma \rhd f_1 \preccurlyeq:: t_1 \quad \cdots \quad \gamma \rhd f_n \preccurlyeq:: t_n}{\gamma \rhd [f_1, \cdots, f_n] \preccurlyeq:: [t_1 \otimes \cdots \otimes t_n]} \text{ [te-seq2]}$$

$$t_1, t_2 \in \{\textbf{string}, [t]\} \frac{\gamma \rhd f_1 \preccurlyeq:: t_1 \quad \gamma \rhd f_2 \preccurlyeq:: t_2}{\gamma \rhd f_1 \boxplus f_2 \preccurlyeq: t_1 \otimes t_2} \text{ [te-p-fcat]}$$

Le typage des termes $[e_1, \cdots, e_n] \sharp f$ est défini par la règle générique [te-match] , en adjonction des règles [te-p-free],[te-p-anti] et autres déjà définies pour les filtres structure.

Enfin, nous devons étendre la fonction $\mathcal{I}nit$ qui calcule un terme par défaut pour tout type t bien formé (cette fonction est utilisée pour la ∇-application).

Définition 6.20 *Fonction de calcul d'un terme par défaut pour un type séquence (extension de 5.7.3)*

$$\mathcal{I}nit([t]) \stackrel{def}{=} [\,]$$

On peut montrer facilement que la propriété de correction $\gamma \rhd t \Rightarrow \gamma \rhd \mathcal{I}nit(t) : t$ (cf 5.13) est préservée.

6.4.4 Sémantique

La priorité des opérandes dans les opération $+, ==$ et $\leq$ reste déterminée par les règles génériques [e-op1] et [e-op2] définies dans la sémantique opérationnelle du noyau. Il reste donc à déterminer la réduction d'un terme séquence ([e-seq]) et la concaténation de deux séquences normalisées ([e-plus-seq])

$$\frac{[\mathcal{S} \vdash e_i] \rightarrow [\mathcal{S} \vdash e_i']}{[\mathcal{S} \vdash [v_1, \cdots, e_i, \cdots, e_n]] \rightarrow [\mathcal{S} \vdash [v_1, \cdots, e_i', \cdots, e_n]]} \text{ [e-seq]}$$

$$[\mathcal{S} \vdash [v_1, \cdots, v_n] + [v_1', \cdots, v_k']] \rightarrow [\mathcal{S} \vdash [v_1, \cdots, v_n, v_1', \cdots, v_k']] \text{ [e-plus-seq]}$$

$$[\mathcal{S} \vdash [v_1, \cdots, v_n] + [\,]] \rightarrow [\mathcal{S} \vdash [v_1, \cdots, v_n]] \text{ [e-plus-seq1]}$$

$$[\mathcal{S} \vdash [\,] + [v_1, \cdots, v_n]] \rightarrow [\mathcal{S} \vdash [v_1, \cdots, v_n]] \text{ [e-plus-seq2]}$$

$$[\mathcal{S} \vdash [v_1, \cdots, \textbf{error}, \cdots, e_n]] \rightarrow [\mathcal{S} \vdash \textbf{error}] \text{ [e-seq-err]}$$

Nous définissons également l'égalité et l'inégalité de deux séquences normalisées par :

$$\frac{[\mathcal{S} \vdash v_1 == v_1'] \to [\mathcal{S} \vdash \textbf{true}] \quad \cdots [\mathcal{S} \vdash v_n == v_n'] \to [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash [v_1, \cdots, v_n] == [v_1', \cdots, v_n']] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-eq-seq]}$$

$$\frac{\exists i \in [1,n] \text{ tel que } \quad [\mathcal{S} \vdash v_i == v_i'] \to [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash [v_1, \cdots, v_n] == [v_1', \cdots, v_n']] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-seq2]}$$

$$\frac{n \neq k}{[\mathcal{S} \vdash [v_1, \cdots, v_n] == [v_1', \cdots, v_k']] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-seq3]}$$

$$\frac{[\mathcal{S} \vdash v_1 <= v_1'] \to [\mathcal{S} \vdash \textbf{true}] \quad \cdots [\mathcal{S} \vdash v_n <= v_n'] \to [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash [v_1, \cdots, v_n] <= [v_1', \cdots, v_{n+k}']] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-ineq-seq]}$$

$$\frac{}{[\mathcal{S} \vdash [\,] <= [v_1, \cdots, v_n]] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-ineq-seq2]}$$

$$\frac{\exists i \in [1,n] \text{ tel que } \quad [\mathcal{S} \vdash v_i <= v_i'] \to [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash [v_1, \cdots, v_n] <= [v_1', \cdots, v_{n+k}']] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-ineq-seq3]}$$

Enfin, le filtrage se fait dans l'ordre des sous-termes.

$$\frac{[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1}] \to [\mathcal{S}^{(1)} \vdash \textbf{true}] \quad \cdots \quad [\mathcal{S}^{(n-1)} \vdash v_n \sharp \boldsymbol{f_n}] \to [\mathcal{S}^{(n)} \vdash \textbf{true}]}{[\mathcal{S} \vdash [v_1, \cdots, v_n] \sharp [\boldsymbol{f_1}, \cdots, \boldsymbol{f_n}]] \to [\mathcal{S}^{(n)} \vdash \textbf{true}]} \text{ [e-match-seq]}$$

$$\frac{[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1}] \to [\mathcal{S}^{(1)} \vdash \textbf{true}] \quad \cdots \quad \begin{array}{c}\exists i \in [1,n] \\ [\mathcal{S}^{(i-1)} \vdash v_i \sharp \boldsymbol{f_i}] \to [\mathcal{S}^{(i)} \vdash \textbf{false}]\end{array}}{[\mathcal{S} \vdash [v_1, \cdots, v_n] \sharp [\boldsymbol{f_1}, \cdots, \boldsymbol{f_n}]] \to [\mathcal{S}^{(i)} \vdash \textbf{false}]} \text{ [e-match-seq2]}$$

$$\frac{n \neq k}{[\mathcal{S} \vdash [v_1, \cdots, v_n] \sharp [\boldsymbol{f_1}, \cdots, \boldsymbol{f_k}]] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-match-seq3]}$$

6.4.5 Correction du système de type

Les termes de type "séquence" respectent la propriété de préservation du type dans les reductions.

Proposition 6.21 *Préservation du type séquence*

$$\begin{array}{l} \gamma \rhd [e_1, \cdots, e_i, \cdots, e_n] : t \\ \Rightarrow \\ \textit{pour tout } \mathcal{S}, \quad \textit{tel que} \quad \Vdash \mathcal{S} : \gamma \textit{ alors} \\ \left\{ \begin{array}{l} [\mathcal{S} \vdash [e_1, \cdots, e_i, \cdots, e_n]] \to [\mathcal{S}' \vdash [e_1, \cdots, e_i', \cdots, e_n]] \\ \gamma \rhd [e_1, \cdots, e_i', \cdots, e_n] : t \\ \Vdash \mathcal{S}' : \gamma \end{array} \right. \end{array}$$

Preuve : Sans difficultés particulières, en utilisant la proposition 4.44 □

Proposition 6.22 *Préservation du type pour l'ensemble des opérations sur les séquences.*

Les opérations de concaténation, de comparaison et de filtrage préservent la relation de typage comme défini en 4.44

Preuve :avec la proposition précédente, et les techniques utilisées dans le chapitre précédent □

6.4.6 Contrôle de type

Nous devons apporter des extensions aux algorithmes proposés dans le noyau du langage.

Définition 6.23 *Algorithme de test syntaxique de convergence (extension de 4.60)*

elif $(e \equiv [e_1, \cdots, e_n])$ *then*
  for $e' \in e_1 \cdots e_n$ **do**
    if conv$(\gamma, e') ==$ **false** *then return* **false**
  done
  return **true**

Définition 6.24 *Algorithme de décision sur la validité des types (extension de 4.64)*

elif $(t \equiv [t'])$ *then*
  return $\mathcal{T}ype(\gamma, t')$

Définition 6.25 *Algorithme de décision sur la relation de sous-typage (extension de 4.65)*

elif $(t_1 \equiv [t_{11}])$ *then*
  if $(t_2 \equiv [t_{21}])$
    return $\mathcal{S}ub(\gamma, t_{11}, t_{21})$
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Définition 6.26 *Algorithme de calcul du type minimal d'un terme (extension de $\mathcal{F}nd$, cf 4.69)*

if $(e \equiv [e_1, \cdots, e_n])$ *then*
  $tt = \bot$
  for $e' \in \{e_1, \cdots, e_n\}$ **do**
    $t' = \mathcal{F}nd(\gamma, e')$
    if $t' \equiv$ **error** *then return* **error**
    $tt = \mathcal{A}ddType(\gamma, tt, t')$
  done
  return $[tt]$ **in** $\{e\}$

Définition 6.27 *Algorithme de calcul du type principal d'un terme séquence (extension de 6.1)*

if $(e \equiv [e_1, \cdots, e_n])$ *then*
  $t = \bot$
  for $e' \in \{e_1, \cdots, e_n\}$ **do**
    $tt = \mathcal{F}ndP(\gamma, e')$
    if $tt \equiv$ **error** *then return* **error**
    $t = \mathcal{A}ddType(\gamma, t, tt)$
  done
  return $[t]$

Le calcul du type principal des filtres séquence est presque identique au précédent. La différence réside dans le fait que l'on calcule l'intersection des types principaux, au lieu de calculer l'union.

Définition 6.28 *Algorithme de calcul du type principal d'un filtre séquence (extension de 6.1)*

if $(e \equiv [f_1, \cdots, f_n])$ *then*
  $t = \top$
  for $f \in \{f_1, \cdots, f_n\}$ **do**
    $tt = \mathcal{F}ndP(\gamma, f)$
    if $tt \equiv$ **error** *then return* **error**
    $t = \mathcal{P}rodType(\gamma, t, tt)$
  done
  return $[t]$

Définition 6.29 *Algorithme de contrôle de conformité (extension de* $\mathcal{C}hk$*, cf 4.6.7)*

if $(e \equiv [e_1, \cdots, e_n])$ *then*
  if $(t \equiv [t_1])$
    for $e' \in \{e_1, \cdots, e_n\}$ **do**
      if $\mathcal{C}hk(\gamma, e', t_1) ==$ **false** *then*
        return **false**
    done
    return **true**
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Définition 6.30 *Algorithme de calcul pour l'opérateur* $\oplus$ *(extension de 4.67)*

elif $(t_1 \equiv [t_{11}])$ *then*
  if $(t_2 \equiv [t_{21}])$ *then*
    return $[\mathcal{A}ddType(\gamma, t_{11}, t_{21})]$
  elif $(t_2 \equiv t$ **in** $S)$ *then*
    if $\mathcal{S}ub(\gamma, t_2, t_1) \equiv$ **true** *then return* t_1
    $t = \bot$
    for $e \in S$**do**
      if $e \equiv [ee_1, \ldots, ee_k]$ *then*
        $tt = \bot$
        for $ee_i \in \{ee_1, \ldots .ee_k\}$**do**
          $tt = \mathcal{A}ddType(\gamma, tt, \mathcal{F}nd(\gamma, ee_i))$
        done
        $t = \mathcal{A}ddType(\gamma, t, [tt])$
      else $t = \top$
    done
    return $\mathcal{A}ddType(\gamma, t_1, t)$
  elif $(t_2 \equiv \bot)$ *then return* t_1
  return $\top$

Définition 6.31 *Algorithme de calcul pour l'opérateur* $\otimes$ *(extension de 4.68)*

elif $(t_1 \equiv [t_{11}])$ *then*
  if $(t_2 \equiv [t_{21}])$ *then*
    return $[\mathcal{P}rodType(\gamma, t_{11}, t_{21})]$
  if $(t_2 \equiv \top)$ *then return* t_1
  return $\bot$

6.5 Ensembles

Les ensembles sont très proches des séquences, si l'on fait abstraction de l'ordre des éléments. Comme ils peuvent comprendre plusieurs éléments identiques, il serait plus rigoureux de les appeller "multi-ensembles" (*multisets, bags*). En fait, ce type est si rigoureusement similaire au type séquence, qu'il serait stupide de dupliquer toutes les définitions et résultats produits dans la section précédente. Il suffit de renommer les règles [te-seq-*] en [te-set-*] et d'adapter les algorithmes. Nous ne présenterons donc que la syntaxe des termes et types, puis la sémantique opérationnelle.

6.5.1 Syntaxe

e	::=	$\{e_1, \cdots, e_n\}$	*(multi-)ensemble non ordonné*
e	::=	$\{\ \}$	*ensemble vide*
t	::=	$\{t_1\}$	*Type ensemble*
f	::=	$\{f_1, \cdots, f_n\}$	*filtre pour termes "ensemble"*

6.5.2 Sémantique

$$\frac{[\mathcal{S} \vdash e_i] \rightarrow [\mathcal{S} \vdash e'_i]}{[\mathcal{S} \vdash \{v_1, \cdots, e_i, \cdots, e_n\}] \rightarrow [\mathcal{S} \vdash \{v_1, \cdots, e'_i, \cdots, e_n\}]} \text{ [e-set]}$$

$$[\mathcal{S} \vdash \{v_1, \cdots, v_n\} + \{v'_1, \cdots, v'_k\}] \rightarrow [\mathcal{S} \vdash \{v_1, \cdots, v_n, v'_1, \cdots, v'_k\}] \text{ [e-plus-set]}$$

$$[\mathcal{S} \vdash \{v_1, \cdots, v_n\} + \{\ \}] \rightarrow [\mathcal{S} \vdash \{v_1, \cdots, v_n\}] \text{ [e-plus-set1]}$$

$$[\mathcal{S} \vdash \{\ \} + \{v_1, \cdots, v_n\}] \rightarrow [\mathcal{S} \vdash \{v_1, \cdots, v_n\}] \text{ [e-plus-set2]}$$

$$[\mathcal{S} \vdash \{v_1, \cdots, \textbf{error}, \cdots, e_n\}] \rightarrow [\mathcal{S} \vdash \textbf{error}] \text{ [e-set-err]}$$

Nous définissons également l'égalité et l'inégalité de deux ensembles normalisés. L'égalité est vérifiée lorsque les deux opérandes ont exactement les mêmes éléments, indépendamment de l'ordre. Cela signifie que le nombre d'occurence de chaque élément dans chacun des ensembles est identique. Ceci est formalisé par [e-eq-set] en utilisant une définition récurrente. Un ensemble est inférieur ou égal à un autre ensemble si tous les éléments du premier sont présents dans le second. Cette définition correspond à une sémantique d'inclusion ensembliste. elle permet de nuancer les comparaisons sur les ensembles puisque $A \leq B \wedge B \leq A$ n'est pas équivalent à $A == B$. Dans les deux cas, les éléments sont comparés en utilisant l'égalité.

$$\exists i \in [1,n] \quad \frac{[\mathcal{S} \vdash v_1 == v_i'] \rightarrow [\mathcal{S} \vdash \textbf{true}] \quad [\mathcal{S} \vdash \{v_2, \cdots, v_n\} == \{v_1', \cdots, v_n'\}] \rightarrow [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash \{v_1, v_2, \cdots, v_n\} == \{v_1', \cdots, v_i', \cdots, v_n'\}] \rightarrow [\mathcal{S} \vdash \textbf{true}]} \text{ [e-eq-set]}$$

$$\frac{n \neq k}{[\mathcal{S} \vdash \{v_1, \cdots, v_n\} == \{v_1', \cdots, v_k'\}] \rightarrow [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-set2]}$$

$$[\mathcal{S} \vdash \{\} == \{\}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \text{ [e-eq-set2]}$$

$$\frac{\forall i \in [1,n], \exists j \in [1, n+k] \quad [\mathcal{S} \vdash v_i == v_j'] \rightarrow [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash \{v_1, \cdots, v_n\} <= \{v_1', \cdots, v_{n+k}'\}] \rightarrow [\mathcal{S} \vdash \textbf{true}]} \text{ [e-ineq-set]}$$

$$[\mathcal{S} \vdash \{\} <= \{v_1, \cdots, v_n\}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \text{ [e-ineq-set2]}$$

$$[\mathcal{S} \vdash \{\} <= \{\}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \text{ [e-ineq-set3]}$$

Enfin, le filtrage d'ensembles s'applique sur des termes de même cardinal, indépendamment de l'ordre. L' opérateur de concaténation $\boxplus$ est étendu pour prendre en compte les ensembles de manière similaire aux chaînes de caractères et aux séquences.

$$\frac{\exists i, j \in [1,n] \quad [\mathcal{S} \vdash v_i \sharp \boldsymbol{f_j}] \rightarrow [\mathcal{S}' \vdash \textbf{true}] \quad [\mathcal{S}' \vdash \{v_1, \cdots, v_n\} \sharp \{\boldsymbol{f_1}, \cdots, \boldsymbol{f_n}\}] \rightarrow [\mathcal{S}'' \vdash \textbf{true}]}{[\mathcal{S} \vdash \{v_1, \cdots, v_i, \cdots, v_n\} \sharp \{\boldsymbol{f_1}, \cdots, \boldsymbol{f_j}, \cdots, \boldsymbol{f_n}\}] \rightarrow [\mathcal{S}'' \vdash \textbf{true}]} \text{ [e-match-set]}$$

$$\frac{\neg(\exists i, j \in [1,n] \mid [\mathcal{S} \vdash v_i \sharp \boldsymbol{f_j}] \rightarrow [\mathcal{S}' \vdash \textbf{true}] \ \wedge \ [\mathcal{S}' \vdash \{v_1, \cdots, v_n\} \sharp \{\boldsymbol{f_1}, \cdots, \boldsymbol{f_n}\}] \rightarrow [\mathcal{S}'' \vdash \textbf{true}])}{[\mathcal{S} \vdash \{v_1, \cdots, v_i, \cdots, v_n\} \sharp \{\boldsymbol{f_1}, \cdots, \boldsymbol{f_j}, \cdots, \boldsymbol{f_n}\}] \rightarrow [\mathcal{S} \vdash \textbf{false}]} \text{ [e-match-set2]}$$

Notons, ici encore, que [e-match-set] est loin de fournir un algorithme pour décider le filtrage, car aucune indication n'est donnée sur la façon de développer l'arbre de recherche.

6.6 Tuples

Les tuples sont une structure de données bien connue, offrant une possibilité simple de combiner des types hétérogènes.

e	$::=$	$\langle e_1, \cdots, e_n \rangle$	*tuple ordonné, d'arité* n
t	$::=$	$\langle t_1, \cdots, t_n \rangle$	*type tuple, produit de n types*

6.6.1 Règles de typage

L'égalité de deux structures est typée via la règle générique [te-eq]. Pour les autres opérations les définitions sont les suivantes :

$$\frac{\gamma \rhd t_1 \quad \cdots \gamma \rhd t_n}{\gamma \rhd \langle t_1, \cdots, t_n \rangle} \text{ [t-tuple]}$$

$$\frac{\gamma \rhd t_1 \preccurlyeq t'_1 \quad \cdots \gamma \rhd t_n \preccurlyeq t'_n}{\gamma \rhd \langle t_1, \cdots, t_n \rangle \preccurlyeq \langle t'_1, \cdots, t'_n \rangle} \text{ [st-tuple]}$$

$$\frac{\gamma \rhd e_1 : t_1 \quad \cdots \quad \gamma \rhd e_n : t_n}{\gamma \rhd \langle e_1, \cdots, e_n \rangle : \langle t_1, \cdots, t_n \rangle} \text{ [te-tuple]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: t_1 \quad \cdots \quad \gamma \rhd e_n \preccurlyeq:: t_n}{\gamma \rhd \langle e_1, \cdots, e_n \rangle \preccurlyeq:: \langle t_1, \cdots, t_n \rangle} \text{ [te-p-tuple]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: \langle t_1, \cdots, t_n \rangle \quad \gamma \rhd e_2 \preccurlyeq:: \langle t_1, \cdots, t_n \rangle}{\gamma \rhd e_1 \leq e_2 : \textbf{bool}} \text{ [te-leq-tuple]}$$

6.6.2 Union minimale et intersection maximale

L'union se calcule récursivement, membre à membre.

Définition 6.32 *Union minimale (extension aux types tuple)*

$$
\begin{array}{l}
\textit{Si } t_1 \equiv \langle t_1, \cdots, t_n \rangle \\
\quad \textit{Si } t_2 \equiv \langle t'_1, \cdots, t'_n \rangle \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} \langle t_1 \oplus t'_1, \cdots, \ell_n : t_n \oplus t'_n \rangle \\
\\
\quad \textit{Sinon si } t_2 \equiv \langle t'_1, \cdots, t'_{n+k} \rangle \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} \top \\
\\
\quad \textit{Sinon si } t_2 \equiv t'_1 \textbf{ in } \{e_1, \cdots, e_k\} \\
\quad \left\{ \begin{array}{l}
\textit{si } \gamma \rhd t_2 \preccurlyeq t_1 \textit{ alors } t_1 \oplus t_2 \stackrel{\textit{def}}{=} t_1 \\
\textit{sinon si } \forall i \in [1,k], \left\{ \begin{array}{l} e_i \equiv \langle e_{i1}, \ldots, e_{in} \rangle \\ \gamma \rhd e_{i1} \preccurlyeq : t_{i1} \quad \cdots \quad \gamma \rhd e_{in} \preccurlyeq : t_{in} \end{array} \right. \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} (t_1 \oplus \langle t_{11}, \ldots, t_{1n} \rangle \oplus \cdots \oplus \langle t_{k1}, \ldots, t_{kn} \rangle) \\
\textit{sinon } t_1 \oplus t_2 \stackrel{\textit{def}}{=} \top
\end{array} \right. \\
\quad \textit{Sinon si } t_1 \equiv t'_1 \textbf{ in } S' \textit{ et } t_2 \not\equiv \langle \cdots \rangle \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} t_2 \oplus t_1 \quad (\textit{ce cas est traité précédement}) \\
\\
\quad \textit{Sinon si } t_2 \not\equiv t'_1 \textbf{ in } S' \textit{ et } t_2 \not\equiv \langle \cdots \rangle \textit{ alors} \\
\qquad t_1 \oplus t_2 \stackrel{\textit{def}}{=} \top
\end{array}
$$

L'intersection se fait également membre à membre.

Définition 6.33 *Intersection maximale (extension aux types tuple)*

$$
\begin{array}{l}
\textit{Si } t_1 \equiv \langle t_1, \ldots, t_n \rangle \\
\quad \textit{si } t_2 \equiv \langle t'_1, \ldots, t'_n \rangle \\
\qquad t_1 \otimes t_2 \stackrel{\textit{def}}{=} \\
\qquad \langle t_1 \otimes t'_1, \ldots, t_n \otimes t'_n \rangle \\
\quad \textit{si } t_2 \equiv t \textbf{ in } \{e_1 \cdots e_k\} \\
\qquad \textit{si } S = \{e \in \{e_1, \ldots, e_k\} | \gamma \rhd e : t_1\} \neq \emptyset \quad t_1 \otimes t_2 \stackrel{\textit{def}}{=} t \textbf{ in } S \\
\qquad \textit{sinon } t_1 \otimes t_2 \stackrel{\textit{def}}{=} \bot \\
\\
\quad \textit{sinon } t_1 \otimes t_2 \stackrel{\textit{def}}{=} \bot
\end{array}
$$

Proposition 6.34 *Preservation des propriétés de $\oplus$ et $\otimes$.*

Le calcul de l'union minimale et de l'intersection maximale de deux types quelconques étendu avec la définition 6.32 et 6.33 possède l'ensemble des propriétés établies dans le noyau.

Preuve : en annexe □

6.6.3 filtrage de tuples

La forme du filtre tuple correspond étroitement à la forme des termes, comme pour les autres types.

$$f \quad ::= \quad \langle f_1, \cdots, f_n \rangle \quad \textit{filtre pour termes tuples}$$

Associée à la syntaxe des filtres, il est nécessaire de pouvoir assigner un type principal pour pouvoir utiliser [te-match].

$$\frac{\gamma \rhd f_1 \preccurlyeq:: t_1 \quad \cdots \quad \gamma \rhd f_n \preccurlyeq:: t_n}{\gamma \rhd \langle f_1, \cdots, f_n \rangle \preccurlyeq:: \langle t_1, \cdots, t_n \rangle} \text{ [te-p-ftuple]}$$

6.6.4 Sémantique

Un tuple est évalué membre après membre, dans l'ordre d'occurence.

$$\frac{[\mathcal{S} \vdash e_i] \to [\mathcal{S} \vdash e_i']}{[\mathcal{S} \vdash \langle v_1, \cdots, e_i, \cdots, e_n \rangle] \to [\mathcal{S} \vdash \langle v_1, \cdots, e_i', \cdots, e_n \rangle]} \text{ [e-tuple]}$$

$$[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle] \to [\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle] e - tuple - norm$$

$$[\mathcal{S} \vdash \{v_1, \cdots, \textbf{error}, \cdots, e_n\}] \to [\mathcal{S} \vdash \textbf{error}] \text{ [e-tuple-err]}$$

Nous définissons également l'égalité et l'inégalité de deux tuples normalisés. L'égalité est vérifiée lorsque les deux opérandes ont exactement les mêmes éléments, dans le même ordre. Un tuple est inférieur ou égal à un autre tuple en appliquant la même approche : en comparant élément par élément, dans l'ordre de définition.

$$\frac{[\mathcal{S} \vdash v_1 == v_1'] \to [\mathcal{S} \vdash \textbf{true}] \quad \cdots [\mathcal{S} \vdash v_n == v_n'] \to [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle == \langle v_1', \cdots, v_n' \rangle] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-eq-tuple]}$$

$$i \in [1, n] \quad \frac{[\mathcal{S} \vdash v_1 == v_1'] \to [\mathcal{S} \vdash \textbf{true}] \quad \cdots [\mathcal{S} \vdash v_i == v_i'] \to [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle == \langle v_1', \cdots, v_n' \rangle] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-tuple2]}$$

$$\frac{[\mathcal{S} \vdash v_1 <= v_1'] \to [\mathcal{S} \vdash \textbf{true}] \quad \cdots [\mathcal{S} \vdash v_n <= v_n'] \to [\mathcal{S} \vdash \textbf{true}]}{[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle <= \langle v_1', \cdots, v_n' \rangle] \to [\mathcal{S} \vdash \textbf{true}]} \text{ [e-ineq-tuple]}$$

$$i \in [1, n] \quad \frac{[\mathcal{S} \vdash v_1 <= v_1'] \to [\mathcal{S} \vdash \textbf{true}] \quad \cdots [\mathcal{S} \vdash v_i <= v_i'] \to [\mathcal{S} \vdash \textbf{false}]}{[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle <= \langle v_1', \cdots, v_n' \rangle] \to [\mathcal{S} \vdash \textbf{false}]} \text{ [e-ineq-tuple2]}$$

Comme pour le type structure, le filtrage d'un tuple se fait membre à membre, dans l'ordre,, en propageant le contexte.

$$\frac{[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1}] \to [\mathcal{S}^{(1)} \vdash \textbf{true}] \quad \cdots \quad [\mathcal{S}^{(n-1)} \vdash v_n \sharp \boldsymbol{f_n}] \to [\mathcal{S}^{(n)} \vdash \textbf{true}]}{[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle \sharp \langle \boldsymbol{f_1}, \cdots, \boldsymbol{f_n} \rangle] \to [\mathcal{S}^{(n)} \vdash \textbf{true}]} \text{ [e-match-tuple]}$$

$$\exists i \in [1, n] \quad \frac{[\mathcal{S} \vdash v_1 \sharp \boldsymbol{f_1}] \to [\mathcal{S}^{(1)} \vdash \textbf{true}] \quad \cdots \quad [\mathcal{S}^{(i-1)} \vdash v_i \sharp \boldsymbol{f_i}] \to [\mathcal{S}^{(i)} \vdash \textbf{false}]}{[\mathcal{S} \vdash \langle v_1, \cdots, v_n \rangle \sharp \langle \boldsymbol{f_1}, \cdots, \boldsymbol{f_n} \rangle] \to [\mathcal{S}^{(i)} \vdash \textbf{false}]} \text{ [e-match-tuple2]}$$

6.6.5 Correction du système de type

Les termes de type “tuple” respectent la propriété de préservation du type dans les réductions.

Proposition 6.35 *Préservation du type tuple*

$$\begin{array}{l}\gamma \rhd \langle e_1, \cdots, e_i, \cdots, e_n\rangle : t \\ \Rightarrow \\ \textit{pour tout } \mathcal{S}, \quad \textit{tel que} \quad \Vdash \mathcal{S} : \gamma \textit{ alors} \\ \left\{ \begin{array}{l} [\mathcal{S} \vdash \langle e_1, \cdots, e_i, \cdots, e_n\rangle] \rightarrow [\mathcal{S}' \vdash \langle e_1, \cdots, e'_i, \cdots, e_n\rangle] \\ \gamma \rhd \langle e_1, \cdots, e'_i, \cdots, e_n\rangle : t \\ \Vdash \mathcal{S}' : \gamma \end{array} \right.\end{array}$$

Preuve : Sans difficultés particulières, en utilisant la proposition 4.44 □

Proposition 6.36 *Préservation du type pour l'ensemble des opérations sur les structures.*

Les opérations de comparaison et de filtrage préservent la relation de typage comme défini en 4.44

Preuve :avec la proposition précédente, et les techniques utilisées dans le chapitre précédent □

Et enfin, nous devons étendre la fonction $\mathcal{I}nit$ qui calcule un terme par défaut pour tout type t bien formé (cette fonction est utilisée pour la ∇-application).

Définition 6.37 *Fonction de calcul d'un terme par défaut pour un type tuple (extension de 5.7.3)*

$$\mathcal{I}nit(\langle t_1, \ldots, t_n\rangle) \stackrel{def}{=} \langle \mathcal{I}nit(t_1), \ldots, \mathcal{I}nit(t_n)\rangle$$

On montre sans difficulté que la propriété de correction $\gamma \rhd t \Rightarrow \gamma \rhd \mathcal{I}nit(t) : t$ (cf 5.13) est préservée.

6.6.6 Contrôle de type

Nous devons apporter des extensions aux algorithmes proposés dans le noyau du langage.

Définition 6.38 *Algorithme de test syntaxique de convergence (extension de 4.60)*

elif $(e \equiv \langle e_1, \cdots, e_n\rangle)$ *then*
  for $e' \in e_1 \cdots e_n$ **do**
    if conv(γ, e') == **false** *then return* **false**
  done
  return **true**

Définition 6.39 *Algorithme de décision sur la validité des types (extension de 4.64)*

elif $(t \equiv \langle t_1 \cdots, t_n\rangle)$ *then*
  return $\mathcal{T}ype(\gamma, t_1)$ *and* $\cdots$ *and* $\mathcal{T}ype(\gamma, t_n)$

Définition 6.40 *Algorithme de décision sur la relation de sous-typage (extension de 4.65)*

elif $(t_1 \equiv \langle t_{11}, \cdots, t_{1n}\rangle)$ *then*
  if $(t_2 \equiv \{t_{21}, \cdots, t_{2n}\})$
    return $\mathcal{S}ub(\gamma, t_{11}, t_{21})$ *and* $\cdots$ *and* $\mathcal{S}ub(\gamma, t_{1n}, t_{2n})$
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Définition 6.41 *Algorithme de contrôle de conformité (extension de $\mathcal{C}hk$ 4.6.7)*

if $(e \equiv \langle e_1, \cdots, e_n \rangle)$ *then*
  if $(t \equiv \langle t_1, \cdots, t_n \rangle)$
    for $i \in [1, n]$ **do**
      if $\mathcal{C}hk(\gamma, e_i, t_i) ==$ **false** *then return* **false**
    done
    return **true**
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Dans l'algorithme suivant, comme pour les structures, l'énumération retournée en résultat pourrait être plus simple : $\top$ **in** $\{e\}$ conviendrait. Il faudrait alors s'assurer que le terme e est confluent et qu'il possède un type bien formé.

Définition 6.42 *Algorithme de calcul du type minimal d'un tuple (extension de 4.69)*

if $(e \equiv \langle e_1, \ldots, e_n \rangle)$ *then*
  $t_1 = \mathcal{F}nd(\gamma, e_1); \cdots; t_n = \mathcal{F}nd(\gamma, e_n)$
  if **error** $\in \{t_1, \cdots, t_n\}$ *then return* **error**
  return $\langle t_1, \ldots, : t_n \rangle$ **in** $\{e\}$

Définition 6.43 *Algorithme de calcul du type principal d'un terme (extension de 6.1)*

if $(e \equiv \langle e_1, \ldots, e_n \rangle)$ *then*
  for $e_i \in \{e_1, \cdots, e_n\}$ **do**
    $t_i = \mathcal{F}ndP(\gamma, e_i)$
    if $t \equiv$ **error** *then return* **error**
  done
  return $\langle t_1, \ldots, t_n \rangle$
if $(e \equiv \langle f_1, \ldots, f_n \rangle)$ *then*
  for $f_i \in \{f_1, \cdots, f_n\}$ **do**
    $t_i = \mathcal{F}ndP(\gamma, f_i)$
    if $t \equiv$ **error** *then return* **error**
  done
  return $\langle t_1, \ldots, t_n \rangle$

Définition 6.44 *Algorithme de calcul pour l'opérateur $\oplus$ (extension de 4.67)*

elif $(t_1 \equiv \langle t_{11}, \ldots, t_{1n} \rangle)$ *then*
  if $(t_2 \equiv \langle t_{21}, \ldots, t_{2n} \rangle)$ *then*
    return $\langle \mathcal{A}ddType(\gamma, t_{11}, t_{21}), \ldots, \mathcal{A}ddType(\gamma, t_{1n}, t_{2n}) \rangle$
  if $(t_2 \equiv \langle t_{21}, \ldots, t_{2k} \rangle)$ *then return* $\top$
  if $(t_2 \equiv t$ **in** $S)$ *then*
    if $\mathcal{S}ub(\gamma, t_2, t_1) \equiv$ **true** *then return* t_1
    $t = \bot$
    for $e \in S$ **do**
      if $e \equiv \langle ee_1, \ldots, ee_n \rangle$ *then*
        $tt = \langle \mathcal{F}nd(\gamma, ee_1), \ldots, \mathcal{F}nd(\gamma, ee_n) \rangle$
        $t = \mathcal{A}ddType(\gamma, t, tt)$
      else $t = \top$
    done
    return $\mathcal{A}ddType(\gamma, t_1, t)$
  if $(t_2 \equiv \bot)$ *then*
    return t_1
  return $\top$

Définition 6.45 *Algorithme de calcul pour l'opérateur $\otimes$ (extension de 4.68)*

elif $(t_1 \equiv \langle t_{11}, \ldots, t_{1n} \rangle)$ *then*
  if $(t_2 \equiv \langle t_{21}, \ldots, t_{2n} \rangle)$ *then*
    return $\langle \mathcal{P}rodType(\gamma, t_{11}, t_{21}), \ldots, \mathcal{P}rodType(\gamma, t_{1n}, t_{2n}) \rangle$
  if $(t_2 \equiv \langle t_{21}, \ldots, t_{2k} \rangle)$ *then return* $\bot$
  if $(t_2 \equiv \top)$ *then return* t_1
  return $\bot$

6.7 Dictionnaires

Les dictionnaires sont une structure de donnée bien connue dans les langages comme *Python* et *Java*, offrant une possibilité simple de stocker des données à l'aide d'une clef d'accès unique, et de les retrouver au moyen d'une fonction de hachage performante.

e	::=	$\{e_1 = e'_1, \cdots, e_n = e'_n\}$	*dictionnaire constitué de couples (clef,donnée)*
e	::=	$\{=\}$	*dictionnaire vide*
t	::=	$\{t_1 : t'_1\}$	*Type "dictionnaire" avec clefs de type t_1 et éléments de type t'_1*

6.7.1 Règles de typage

$$\frac{\gamma \rhd t \quad \gamma \rhd t'}{\gamma \rhd \{t : t'\}} \text{ [t-dic]}$$

$$\frac{\gamma \rhd t_1 \preccurlyeq t_2 \quad \gamma \rhd t'_1 \preccurlyeq t'_2}{\gamma \rhd \{t_1 : t'_1\} \preccurlyeq \{t_2 : t'_2\}} \text{ [st-dic]}$$

$$\frac{\gamma \rhd e_1 : t, \gamma \rhd e'_1 : t' \quad \cdots \quad \gamma \rhd e_n : t, \gamma \rhd e'_n : t'}{\gamma \rhd \{e_1 = e'_1, \cdots, e_n = e'_n\} : \{t : t'\}} \text{ [te-dic]}$$

$$\gamma \rhd \{=\} : \{\bot : \bot\} \text{ [te-empty-dic]}$$

$$\frac{\gamma \rhd e_1 : \{t : t'\} \quad \gamma \rhd e_2 : \{t : t'\}}{\gamma \rhd e_1 + e_2 : \{t : t'\}} \text{ [te-plus-dic]}$$

$$t, t' \neq \top \quad \frac{\gamma \rhd e_1 : \{t : t'\} \quad \gamma \rhd e_2 : \{t : t'\}}{\gamma \rhd e_1 \leq e_2 : \textbf{bool}} \text{ [te-leq-dic]}$$

Nous précisons le typage principal des termes "dictionnaire" de la manière suivante.

Proposition 6.46 *Typage principal d'un terme "dictionnaire"*

$$\frac{\gamma \rhd e_1 \preccurlyeq:: t_1, \gamma \rhd e'_1 \preccurlyeq:: t'_1 \quad \cdots \quad \gamma \rhd e_n \preccurlyeq:: t_n, \gamma \rhd e'_n \preccurlyeq:: t'_n}{\gamma \rhd \{e_1 = e'_1, \cdots, e_n = e'_n\} \preccurlyeq:: \{(t_1 \oplus \cdots \oplus t_n) : (t'_1 \oplus \cdots \oplus t'_n)\}} \text{ [*te-p-dic*]}$$

$$\gamma \rhd \{=\} \preccurlyeq:: \{\bot : \bot\} \text{ [*te-p-dic2*]}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: \{t_1 : t'_1\} \quad \gamma \rhd e_2 \preccurlyeq:: \{t_2 : t'_2\}}{\gamma \rhd e_1 + e_2 \preccurlyeq:: \{(t_1 \oplus t_2) : (t'_1 \oplus t'_2)\}} \text{ [*te-p-plus-dic*]}$$

6.7.2 Union minimale et intersection maximale

Définition 6.47 *Union minimale (extension aux types dictionnaire)*

if $t_1 \equiv \{t_1 : t'_1\}$

$\quad$ *if* $t_2 \equiv \{t_2 : t'_2\} \quad t_1 \oplus t_2 \stackrel{def}{=} \{(t_1 \oplus t_2) : (t'_1 \oplus t'_2)\}$

$\quad$ *if* $t_2 \equiv t'_1$ **in** $\{e_1, \cdots, e_n\}$

$\quad\quad$ *si* $\gamma \rhd t_2 \preccurlyeq t_1$ *alors* $t_1 \oplus t_2 \stackrel{def}{=} t_1$

$\quad\quad$ *sinon si* $\forall i \in [1,n], \left\{ \begin{array}{l} e_i \equiv \{e_{i1} = e'_{i1}, \ldots, e_{ik} = e'_{ik}\} \\ \gamma \rhd e_{i1} \preccurlyeq: t_{i1}, \gamma \rhd e'_{i1} \preccurlyeq: t'_{i1} \quad \cdots \quad \gamma \rhd e_{ik} \preccurlyeq: t_{ik}, \gamma \rhd e'_{ik} \preccurlyeq: t'_{ik} \end{array} \right.$

$\quad\quad\quad t_1 \oplus t_2 \stackrel{def}{=} (t_1 \oplus \{(t_{11} \oplus \cdots t_{1k}) : (t'_{11} \oplus \cdots t'_{1k})\} \oplus \cdots \oplus \{(t_{n1} \oplus \cdots t_{nk}) : (t'_{n1} \oplus \cdots t'_{nk})\})$

$\quad\quad$ *sinon* $t_1 \oplus t_2 \stackrel{def}{=} \top$

Sinon si $t_1 \equiv t'_1$ **in** S' *et* $t_2 \not\equiv [t]$

$\quad t_1 \oplus t_2 \stackrel{def}{=} t_2 \oplus t_1$ (*ce cas est traité précédement*)

Sinon si $t_2 \not\equiv t'_1$ **in** S' *et* $t_2 \not\equiv [t]$ *alors*

$\quad t_1 \oplus t_2 \stackrel{def}{=} \top$

Définition 6.48 *Intersection maximale (extension au type dictionnaire)*

Si $t_1 \equiv \{t_{11} : t'_{11}\}$

$\quad$ *si* $t_2 \equiv \{t_{21} : t'_{21}\} \quad t_1 \otimes t_2 \stackrel{def}{=} \{(t_{11} \otimes t_{21}) : (t'_{11} \otimes t'_{21})\}$

$\quad$ *si* $t_2 \equiv t$ **in** $\{e_1 \cdots e_n\}$

$\quad\quad$ *si* $S = \{e \in \{e_1, \ldots, e_n\} | \gamma \rhd e : t_1\} \neq \emptyset \quad t_1 \otimes t_2 \stackrel{def}{=} t$ **in** S

$\quad\quad$ *sinon* $t_1 \otimes t_2 \stackrel{def}{=} \bot$

$\quad$ *sinon* $t_1 \otimes t_2 \stackrel{def}{=} \bot$

Proposition 6.49 *Préservation des propriétés de* $\oplus$ *et* $\otimes$.

Le calcul de l'union minimale et de l'intersection maximale de deux types quelconques étendu avec les définitions 6.47 et 6.48 possède l'ensemble des propriétés établies dans le noyau.

Preuve : en annexe

□

6.7.3 filtrage des dictionnaires

Enfin, les filtres dictionnaires sont tels que la clef est un terme. Ainsi, on ne peut pas appliquer d' opérations de filtrage sur les clefs d'un dictionnaire, mais seulement sur ses éléments.

$$f \quad ::= \quad \{e_1 = f_1, \cdots, e_n = f_n\} \quad \textit{filtre pour termes "dictionnaires"}$$

$$\frac{\gamma \rhd e_1 \preccurlyeq:: t_1, \gamma \rhd f_1 \preccurlyeq:: t'_1 \quad \cdots \quad \gamma \rhd e_n \preccurlyeq:: t_n, \gamma \rhd f_n \preccurlyeq:: t'_n}{\gamma \rhd \{e_1 = f_1, \ldots, e_n = f_n\} \preccurlyeq:: \{(t_1 \oplus \cdots \oplus t_n) : (t'_1 \otimes \cdots \otimes t'_n)\}} \text{ [te-p-fdic]}$$

Notons, ici encore, que le typage des termes $e \sharp f$ est assuré par la règle "générique" [te-match].

6.7.4 Sémantique

Les règles suivantes décrivent la façon dont un terme dictionnaire est réduit : chaque couple (clef, élément) est traité dans l'ordre d'occurence. Pour chacun d'eux, la clef est d'abord calculée ([e-dic] s'applique tant que la clef n'est pas sous forme normale). Si une clef devient normale et de plus, si elle est déja utilisée comme clef dans le dictionnaire, alors le nouvel élément se substitut à l'ancien ([e-dic2]). Enfin, l'élément associé est réduit ([e-dic3]) pas par pas. Un dictionnaire dont les clefs et les éléments sont sous forme normale est lui-même normal [e-norm-dic].

[e-dic]

$$\frac{[\mathcal{S} \vdash e_i] \to [\mathcal{S} \vdash e_i'']}{[\mathcal{S} \vdash \{v_1 = v_1', \cdots, e_i = e_i', \cdots, e_n = e_n'\}] \to [\mathcal{S} \vdash \{v_1 = v_1', \cdots, e_i'' = e_i', \cdots, e_n = e_n'\}]}$$

[e-dic2]

$$\frac{[\mathcal{S} \vdash e_i] \to_{\circ} \ [\mathcal{S} \vdash v_i] \quad [\emptyset \vdash v_i == v_k] \to [\emptyset \vdash \textbf{true}]}{[\mathcal{S} \vdash \{\cdots, v_k = v_k', \cdots, e_i = e_i', \cdots\}] \to [\mathcal{S} \vdash \{\cdots, v_k = e_i', \cdots\}]}$$

[e-dic3]

$$\frac{[\mathcal{S} \vdash e_i'] \to [\mathcal{S} \vdash e_i'']}{[\mathcal{S} \vdash \{v_1 = v_1', \cdots, v_i = e_i', \cdots, e_n = e_n'\}] \to [\mathcal{S} \vdash \{v_1 = v_1', \cdots, v_i = e_i'', \cdots, e_n = e_n'\}]}$$

$$[\mathcal{S} \vdash \{v_1 = v_1', \cdots, v_n = v_n'\}] \to [\mathcal{S} \vdash \{v_1 = v_1', \cdots, v_n = v_n'\}] \ \text{[e-norm-dic]}$$

$$[\mathcal{S} \vdash \{v_1 = v_1', \cdots, \textbf{error} = e_i', \cdots, e_n = e_n'\}] \to [\mathcal{S} \vdash \textbf{error}] \ \text{[e-dic-err1]}$$

$$[\mathcal{S} \vdash \{v_1 = v_1', \cdots, v_i = \textbf{error}, \cdots, e_n = e_n'\}] \to [\mathcal{S} \vdash \textbf{error}] \ \text{[e-dic-err2]}$$

La concaténation de deux dictionnaires est plus complexe que pour les séquences et les ensembles, car il peut exister des clefs en commun, et chaque clef doit rester unique après l'opération. La règle [e-cat-dic] montre que l'opérande droite est prioritaire lorsqu'il existe des clefs en commun.

[e-cat-dic]

$$\begin{array}{l} \{ \quad v_{11} = v_{11}', \ldots, v = v', \ldots, v_{1n} = v_{1n}' \quad \} \\ \qquad + \\ \{ \quad v_{21} = v_{21}', \ldots, v = v'', \ldots, v_{2k} = v_{2k}' \quad \} \end{array} \to \begin{array}{l} \{ \quad v_{11} = v_{11}', \ldots, v = v'', \ldots, v_{1n} = v_{1n}', \\ \qquad v_{21} = v_{21}', \ldots\ldots, v_{2k} = v_{2k}' \qquad \} \end{array}$$

[e-cat-dic2]

$$\begin{array}{l} \{ \quad v_{11} = v_{11}', \ldots, v_{1n} = v_{1n}' \quad \} \\ \qquad + \\ \{ \quad v_{21} = v_{21}', \ldots, v_{2k} = v_{2k}' \quad \} \end{array} \to \begin{array}{l} \{ \quad v_{11} = v_{11}', \ldots, v_{1n} = v_{1n}', \\ \qquad v_{21} = v_{21}', \ldots, v_{2k} = v_{2k}' \quad \} \end{array}$$

$$[\mathcal{S} \vdash \{v_{11} = v_{11}', \ldots, v_{1n} = v_{1n}'\} + \{=\}] \to [\mathcal{S} \vdash \{v_{11} = v_{11}', \ldots, v_{1n} = v_{1n}'\}] \ \text{[e-cat-void]}$$

$$[\mathcal{S} \vdash \{=\} + \{v_{11} = v_{11}', \ldots, v_{1n} = v_{1n}'\}] \to [\mathcal{S} \vdash \{v_{11} = v_{11}', \ldots, v_{1n} = v_{1n}'\}] \ \text{[e-cat-void2]}$$

Nous définissons à présent l'égalité et l'inégalité de deux dictionnaires normalisés. L'égalité est vérifiée lorsque les deux opérandes ont exactemment les mêmes éléments (l'ordre reste quelconque, cf [e-eq-dic], et les comparaisons des clefs et éléments utilisent $==$).

$$\frac{\forall i \in [1,n] \quad \exists j \in [1,n] | ([\mathcal{S} \vdash v_{1i} == v_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \ \wedge \ [\mathcal{S} \vdash v'_{1i} == v'_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{true}])}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} == \{v_{21} = v'_{21}, \cdots, v_{2n} = v'_{2n}\}] \rightarrow [\mathcal{S} \vdash \textbf{true}]} \text{ [e-eq-dic]}$$

$$\frac{\begin{array}{c}\forall i \in [1,n] \\ \exists j \in [1,n] | ([\mathcal{S} \vdash v_{1i} == v_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \ \wedge \ [\mathcal{S} \vdash v'_{1i} == v'_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{false}]) \\ \vee \\ \forall j \in [1,n] | ([\mathcal{S} \vdash v_{1i} == v_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{false}])\end{array}}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} == \{v_{21} = v'_{21}, \cdots, v_{2n} = v'_{2n}\}] \rightarrow [\mathcal{S} \vdash \textbf{false}]} \text{ [e-eq-dic2]}$$

Un dictionnaire est inférieur ou égal à un autre dictionnaire lorsque toutes les clefs du premier sont définies dans le second, et que les éléments associés satisfont eux-même la relation $\leq$. Les règles [e-eq-dic2] et [e-ineq-dic2] sont les complémentaires "négatifs", et sont donc exclusives par rapport à leurs symétriques [e-eq-dic] et [e-ineq-dic].

$$\frac{n \leq k \quad \forall i \in [1,n], \ \exists j \in [1,k] \quad ([\mathcal{S} \vdash v_{1i} == v_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \ \wedge \ [\mathcal{S} \vdash v'_{1i} <= v'_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{true}])}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} <= \{v_{21} = v'_{21}, \cdots, v_{2k} = v'_{2k}\}] \rightarrow [\mathcal{S} \vdash \textbf{true}]} \text{ [e-ineq-dic]}$$

$$\frac{\begin{array}{l}n \leq k \\ \forall i \in [1,n], \quad \begin{array}{c}\exists j \in [1,k] | ([\mathcal{S} \vdash v_{1i} == v_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{true}] \ \wedge \ [\mathcal{S} \vdash v'_{1i} <= v'_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{false}]) \\ \vee \\ \forall j \in [1,n] | ([\mathcal{S} \vdash v_{1i} == v_{2j}] \rightarrow [\mathcal{S} \vdash \textbf{false}])\end{array}\end{array}}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} <= \{v_{21} = v'_{21}, \cdots, v_{2k} = v'_{2k}\}] \rightarrow [\mathcal{S} \vdash \textbf{false}]} \text{ [e-ineq-dic2]}$$

La sémantique du filtrage de dictionnaires ressemble à celle des structures. Toutefois, les cardinaux doivent être identiques, et les clefs sont comparées par évaluation de l'égalité.

[e-match-dic]
$$\frac{\forall i \in [1,n],\ \exists j \in [1,n] \mid [\emptyset \vdash v_{2i} == v_{1j}] \rightarrow [\emptyset \vdash \textbf{true}] \quad [\mathcal{S}^{(i)} \vdash v_{1j} \sharp \boldsymbol{f_i}] \rightarrow [\mathcal{S}^{(i+1)} \vdash \textbf{true}]}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} \sharp \{v_{21} = \boldsymbol{f_1}, \cdots, v_{2n} = \boldsymbol{f_n}\}] \rightarrow [\mathcal{S}^{(n)} \vdash \textbf{true}]}$$

[e-match-dic2]
$$\frac{\exists i \in [1,n],\ \mid\ \forall j \in [1,n]\ [\emptyset \vdash v_{2i} == v_{1j}] \rightarrow [\emptyset \vdash \textbf{false}]}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} \sharp \{v_{21} = \boldsymbol{f_1}, \cdots, v_{2n} = \boldsymbol{f_n}\}] \rightarrow [\mathcal{S} \vdash \textbf{false}]}$$

[e-match-dic3]
$$\frac{\forall i \in [1,n],\ \exists j \in [1,n] \mid [\emptyset \vdash v_{2i} == v_{1j}] \rightarrow [\emptyset \vdash \textbf{true}] \quad [\mathcal{S}^{(i)} \vdash v_{1j} \sharp \boldsymbol{f_i}] \rightarrow [\mathcal{S}^{(i+1)} \vdash \textbf{false}]}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} \sharp \{v_{21} = \boldsymbol{f_1}, \cdots, v_{2n} = \boldsymbol{f_n}\}] \rightarrow [\mathcal{S}^{(i+1)} \vdash \textbf{false}]}$$

[e-match-dic3]
$$\frac{n \neq k}{[\mathcal{S} \vdash \{v_{11} = v'_{11}, \cdots, v_{1n} = v'_{1n}\} \sharp \{v_{2k} = \boldsymbol{f_k}, \cdots, v_{2k} = \boldsymbol{f_k}\}] \rightarrow [\mathcal{S} \vdash \textbf{false}]}$$

6.7.5 Correction du système de type

Les termes de type “dictionnaire” respectent la propriété de préservation du type dans les réductions.

Proposition 6.50 *Préservation du type dictionnaire*

$$\begin{array}{l}\gamma \rhd \{e_1 = e'_1, \cdots, e_n = e'_n\} : t \\ \Rightarrow \\ \textit{pour tout } \mathcal{S}, \quad \textit{tel que} \quad \Vdash \mathcal{S} : \gamma \ \textit{alors} \\ \left\{ \begin{array}{l} [\mathcal{S} \vdash \{e_1 = e'_1, \cdots, e_n = e'_n\}] \rightarrow [\mathcal{S}' \vdash e] \\ \gamma \rhd e : t \\ \Vdash \mathcal{S}' : \gamma \end{array} \right.\end{array}$$

Preuve : Sans difficultés particulières, en utilisant la proposition 4.44 □

Proposition 6.51 *Préservation du type pour l’ensemble des opérations sur les dictionnaires.*

Les opérations de concaténation, de comparaison et de filtrage préservent la relation de typage comme défini en 4.44

Preuve :avec la proposition précédente, et les techniques utilisées dans le chapitre précédent □

Et enfin, nous devons étendre la fonction $\mathcal{I}nit$ qui calcule un terme par défaut pour tout type t bien formé (cette fonction est utilisée pour la ∇-application).

Définition 6.52 *Fonction de calcul d’un terme par défaut pour un type dictionnaire (extension de 5.7.3)*

$$\mathcal{I}nit(\{t_1 : t_2\}) \stackrel{def}{=} \{=\}$$

On montre sans difficulté que la propriété de correction $\gamma \ \rhd \ t \Rightarrow \gamma \rhd \mathcal{I}nit(t) : t$ (cf 5.13) est préservée.

6.7.6 Contrôle de type

Nous devons apporter des extensions aux algorithmes proposés dans le noyau du langage.

Définition 6.53 *Algorithme de test syntaxique de convergence (extension de 4.60)*

elif $(e \equiv \{e_1 = e'_1, \cdots, e_n = e'_n\})$ *then*
  for $e' \in e_1 \cdots e_n$ **do**
    if conv$(\gamma, e') ==$ **false** *then return* **false**
  done
  for $e' \in e'_1 \cdots e'_n$ **do**
    if conv$(\gamma, e') ==$ **false** *then return* **false**
  done
  return **true**

Définition 6.54 *Algorithme de décision sur la validité des types (extension de 4.64)*

elif $(t \equiv \{t_1 : t_2\})$ *then*
  return $\mathcal{T}ype(\gamma, t_1)$ *and* $\cdots$ *and* $\mathcal{T}ype(\gamma, t_2)$

Définition 6.55 *Algorithme de décision sur la relation de sous-typage (extension de 4.65)*

elif $(t_1 \equiv \{t_{11} : t_{12}\})$ *then*
  if $(t_2 \equiv \{t_{21} : t_{22}\})$
    return $\mathcal{S}ub(\gamma, t_{11}, t_{21})$ *and* $\mathcal{S}ub(\gamma, t_{12}, t_{22})$
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Définition 6.56 *Algorithme de contrôle de conformité (extension de $\mathcal{C}hk$ 4.6.7)*

if $(e \equiv \{e_1 = e'_1, \cdots, e_n = e'_n\})$ *then*
  if $(t \equiv \{t_1 : t_2\})$
    for $i \in [1, n]$ **do**
      if $\mathcal{C}hk(\gamma, e_i, t_1) ==$ **false** *then return* **false**
      if $\mathcal{C}hk(\gamma, e'_i, t_2) ==$ **false** *then return* **false**
    done
    return **true**
  elif $(t_2 \equiv \top)$ *then*
    return **true**
  else return **false**

Dans l'algorithme suivant, l'énumération retournée en résultat pourrait être plus simple : $\top$ **in** $\{e\}$ conviendrait. Il faudrait alors s'assurer que le terme e est confluent et qu'il possède un type bien formé.

Définition 6.57 *Algorithme de calcul du type minimal d'un dictionnaire (extension de 4.69)*

if $(e \equiv \{e_1 = e'_1, \ldots, e_n = e'_n\})$ *then*
  $t_1 = \bot; t_2 = \bot$
  for $e' \in \{e_1, \ldots, e_n\}$**do**
    $t = \mathcal{F}nd(\gamma, e')$
    if $t \equiv$ **error** *then return* **error**
    $t_1 = \mathcal{A}ddType(\gamma, t_1, t)$
  done
  for $e' \in \{e'_1, \ldots, e'_n\}$**do**
    $t = \mathcal{F}nd(\gamma, e')$
    if $t \equiv$ **error** *then return* **error**
    $t_2 = \mathcal{A}ddType(\gamma, t_2, t)$
  done
  return $\{t_1 : t_2\}$ **in** $\{e\}$

Définition 6.58 *Algorithme de calcul du type principal d'un dictionnaire (extension de 6.1)*

if $(e \equiv \{e_1 = e'_1, \ldots, e_n = e'_n\})$ *then*
  $t_1 = \bot; t_2 = \bot$
  for $e_i \in \{e_1, \cdots, e_n\}$ **do**
    $tt = \mathcal{F}ndP(\gamma, e_i)$
    if $tt \equiv$ **error** *then return* **error**
    $t_1 = \mathcal{A}ddType(\gamma, t_1, tt)$
  done
  for $e'_i \in \{e'_1, \cdots, e'_n\}$ **do**
    $tt = \mathcal{F}ndP(\gamma, e'_i)$
    if $tt \equiv$ **error** *then return* **error**
    $t_2 = \mathcal{A}ddType(\gamma, t_2, tt)$
  done
  return $\{t_1 : t_2\}$
if $(e \equiv \{e_1 = f_1, \ldots, e_n = f_n\})$ *then*
  $t_1 = \bot; t_2 = \top$
  for $e_i \in \{e_1, \cdots, e_n\}$ **do**
    $tt = \mathcal{F}ndP(\gamma, e_i)$
    if $tt \equiv$ **error** *then return* **error**
    $t_1 = \mathcal{A}ddType(\gamma, t_1, tt)$
  done
  for $f_i \in \{f_1, \cdots, f_n\}$ **do**
    $tt = \mathcal{F}ndP(\gamma, f_i)$
    if $tt \equiv$ **error** *then return* **error**
    $t_2 = \mathcal{P}rodType(\gamma, t_2, tt)$
  done
  return $\{t_1 : t_2\}$

Définition 6.59 *Algorithme de calcul pour l'opérateur $\oplus$ (extension de 4.67)*

elif $(t_1 \equiv \{t_{11} : t_{12}\})$ *then*
  if $(t_2 \equiv \{t_{21} : t_{22}\})$ *then*
    return $\{\mathcal{A}ddType(\gamma, t_{11}, t_{21}) : \mathcal{A}ddType(\gamma, t_{12}, t_{22})\}$
  if $(t_2 \equiv t$ **in** $S)$ *then*
    if $\mathcal{S}ub(\gamma, t_2, t_1) \equiv$ **true** *then return* t_1
    for $e \in S$ **do**
      if $e \equiv \{ee_1 = ee'_1, \ldots, ee_n = ee'_n\}$ *then*
      $t_1 = \bot; t_2 = \bot$
        for $ee \in \{ee_1, \ldots, ee_n\}$ **do**
            $t_1 = \mathcal{A}ddType(\gamma, t_1, \mathcal{F}nd(\gamma, ee))$
        done
        for $ee' \in \{ee'_1, \ldots, ee'_n\}$ **do**
            $t_2 = \mathcal{A}ddType(\gamma, t_2, \mathcal{F}nd(\gamma, ee'))$
        done
        $t = \mathcal{A}ddType(\gamma, t, \{t_1 : t_2\})$
      else $t = \top$
    done
    return $\mathcal{A}ddType(\gamma, t_1, t)$
  if $(t_2 \equiv \bot)$ *then*
    return t_1
  return $\top$

Définition 6.60 *Algorithme de calcul pour l'opérateur $\otimes$ (extension de 4.68)*

elif $(t_1 \equiv \{t_{11} : t_{12}\})$ *then*
  if $(t_2 \equiv \{t_{21} : t_{22}\})$ *then*
    return $\{\mathcal{P}rodType(\gamma, t_{11}, t_{21}) : \mathcal{P}rodType(\gamma, t_{11}, t_{22})\}$
  if $(t_2 \equiv \top)$ *then return* t_1
  return $\bot$

6.8 Synthèse

L'introduction de types de données structurés a requis un traitement plus fin de la notion de type minimal, à la fois au niveau formel et algorithmique. De plus, une nouvelle relation de typage est venue enrichir la palette d'outils logique pour raisonner sur les propriétés du système de type : le type principal d'un terme ou d'un filtre. Cette notion permet de répondre aux difficultés soulevées par la nature duale d'un filtre qui peut comporter des variables libres susceptibles d'être modifiées et des constantes ou variables devant simplement être comparées lors de l'appariement. Le type principal est calculable au moyen d'un algorithme dont nous avons fourni la description détaillée tout au long de l'étude des cinq types composites que nous avons conduit dans ce chapitre. Ces derniers, importants du point de vue de l'expressivité, ont pu être intégrés avec succés au système de type, à la syntaxe et à la sémantique opérationnelle. La propriété fondamentale de préservation du type lors des réductions a été établie pour l'ensemble des nouveaux types proposés. Les algorithmes permettant leur utilisation réelle via un contrôleur de type ont été proposés. Toutefois, il n'a pas été possible dans le cadre de ce travail de développer la caractérisation mathématique de ces algorithmes aussi loin que la rigueur l'aurait exigée. Par exemple, la correction et la terminaison de l' ensemble du contrôle de type n'a pu être établie, non de par la difficulté de la preuve en elle-même, mais de par le volume de travail demandé. Cependant, la structure des algorithmes et la façon dont ils ont été incrémentalement étendus ne laisse pas entrevoir de problèmes particulier quant à leur correction et terminaison [1].

[1] Notons encore que ces algorithmes ont été codés et largement testés.

Compositions syntaxiques

7.1 Introduction

Ce chapitre se propose de conclure la partie théorique de cette étude sur l'apport fondamental de nos machines abstraites polymorphes : la composition syntaxique. Nous avions évoqué dans l'état de l'art l'importance de pouvoir maximiser la réutilisation de composants, généralement difficiles à concevoir, écrire et tester, surtout dans le domaine applicatif que nous avons considéré. Lorsque l'on cherche à composer *syntaxiquement* du code, c'est-à-dire en quelque sorte à "mélanger" des *map* afin de produire de nouvelles actions à partir d'actions existentes, il faut résoudre trois grands problèmes :

1. définir la nouvelle signature ou encore calculer le nouveau type

   Plus la méthode est souple, meilleur sera le polymorphisme et donc la réutilisation. Surtout, le polymorphisme laisse entrevoir une réutilisation qui minimise l'effort a priori, ce qui n'est pas facile à obtenir même, par exemple, avec les technologies à objets. D'autre part, la recherche de souplesse ne doit pas faire disparaître les garanties vis à vis d'une exécution correcte qu'apporte un contrôle de type statique.
2. définir la composition des contextes
   C'est une considération statique et opérationnelle, qui aura des implications sur la façon dont le programmeur percevra(et utilisera) les opérateurs de composition.
3. définir (et exprimer) la sémantique de la composition
   Elle doit ètre non seulement précise et cohérente, mais clairement perçue, une fois encore, par le programmeur.

Pour le premier point, il est intéressant, du point de vue du système de type, de calculer l'équivalent du plus grand sous-type commun $\otimes$ ("intersection ensembliste"), car contrairement à son dual $\oplus$, cette opération permet de conserver le type le plus spécialisé, et donc celui sur lequel nous avons le plus d'information (par exemple, peu d'opérations sont disponibles sur des termes de type $\top$). Par exemple, si une *map* m_1 utilise un paramètre formel x en entrée, et une autre *map* m_2 utilise y de la même manière, il est nécessaire qu'une nouvelle machine m_3 réunissant les actions de m_1 et m_2 puisse travailler sur un paramètre z qui soit compatible avec les actions de m_1 et de m_2. Si m_1 calcule l'inverse d'une liste quelconque x (x a pour type $[\top]$) et m_2 est capable de calculer la concaténation d'une liste de chaînes y (y a pour type [**string**]), alors il est légitime d'attendre que m_3 soit capable par exemple de renverser une liste de chaînes puis de les concaténer (z aurait alors le type $[\top] \otimes$ [**string**] $=$ [**string**]). Toutefois, si la composition de deux map $m : t_1 \Longrightarrow t_2$ et $m' : t'_1 \Longrightarrow t'_2$ se calcule par $m \star m' : t_1 \otimes t'_1 \Longrightarrow t_2 \otimes t'_2$, nous n'avons aucune garantie que la nouvelle machine $m \star m'$ possède bien le type requis, à savoir $t_1 \otimes t'_1 \Longrightarrow t_2 \otimes t'_2$. Par exemple $x = 10$ est bien typé dans un contexte où x : **num**. Par contre, si l'on compose séquentiellement cette première instruction avec $x = x +' s'$, bien typée dans un contexte où x : **string**, on obtient $x = 10; x = x +' s'$ dans un contexte x : **num** $\otimes$ **string** qui s'écrit encore $x : \bot$. Il est évident dans ce cas que la composition n'est pas bien typée. Ainsi il reste nécessaire de vérifier qu'une composition est bien typée par rapport au nouveau type calculé.

Le second point est intimement lié au premier : dans une machine abstraite, une partie du contexte est représenté par les paramètres d'entrée et de sortie, et l'autre par les déclarations de variables locales à la machine (il n'est pas question ici des variables internes aux systèmes d'action). Nos opérateurs de composition devront donc fixer également comment les variables locales sont traitées et les conflits détectés et/ou résolus.

Le troisième point est bien sûr fondamental. Dans les langages à objets, la composition se fait par héritage simple ou multiple et par surcharge. Donc, in fine, l'unité de composition est la méthode et nous

avons globalement une sémantique d'union ensembliste. Dans Circus, nous proposons comme unité de composition les systèmes d'action, collections ordonnées de règles ou autres actions impératives, et comme sémantique des connecteurs logiques conditionnels (**Then** , **Else**) ou impératifs (**Before**). La différence est que la sémantique est plus finement contrôlée, en ce sens qu'elle exprime un comportement à l'exécution (ce qui n'est pas le cas dans le modèle à objets). Le premier résultat direct concerne la réutilisation, où le code permettant de faire la "glue" entre les composants est minimisé. Le second concerne le typage qui est capable de calculer et de propager les riches informations de types offertes par notre système de contrôle. L'opérateur de composition le plus original est l'itérateur, noté $\circledast(m)$ qui transforme toute machine en une machine qui itère ses traitements internes. La conception d'un tel opérateur se justifie par la remarque que la boucle itérative "classique" est le principal obstacle à la composition, car elle ne peut pas être "brisée" pour accéder ou modifier des traitements internes. Avec cette approche, le programmeur est incité à "externaliser" ses traitements itératifs : il n'écrit que les opérations essentielles, car il dispose ultérieurement de $\circledast()$ pour généraliser son traitement. En retour, le code non-itératif est composable au sein d'autres traitements ou recombinable dans des itérations plus vastes.

7.2 Actions impératives

Dans une première étape, nous enrichissons les opérations impératives avec deux nouveaux connecteurs

7.2.1 Syntaxe

Le premier groupe de définition porte sur des opérateurs qui permettent de “connecter” des actions impératives en fonction du résultat de leur évaluation (**none** ou **unit**) . Ils sont donc bien adaptés à la composition de règles, de la forme $h \Rightarrow e$, ou de systèmes d'actions.

$$\begin{array}{lcll} e & ::= & e_1 \textbf{ then } e_2 & \textit{évaluer } e_2 \textit{ si } e_1 \textit{ est évalué à } \textbf{unit} \\ e & ::= & e_1 \textbf{ else } e_2 & \textit{évaluer } e_2 \textit{ si } e_1 \textit{ est évalué à } \textbf{none} \end{array}$$

7.2.2 Sémantique

Elle est très simplement définie, grâce une fois encore au test. Notons également ici que le contexte pourra être modifié lors de l'évaluation de la condition.

$$[\mathcal{S} \vdash e_1 \textbf{ then } e_2] \rightarrow [\mathcal{S} \vdash \textbf{if } e_1 == \textbf{unit then } e_2 \textbf{ else none}] \quad \text{[e-then]}$$

$$[\mathcal{S} \vdash e_1 \textbf{ else } e_2] \rightarrow [\mathcal{S} \vdash \textbf{if } e_1 == \textbf{none then } e_2 \textbf{ else unit}] \quad \text{[e-else]}$$

7.2.3 Typage

Le typage est sans surprise :

$$\frac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 : \textbf{Unit}}{\gamma \rhd e_1 \textbf{ then } e_2 : \textbf{Unit}} \text{ [te-then]} \qquad \frac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 : \textbf{Unit}}{\gamma \rhd e_1 \textbf{ else } e_2 : \textbf{Unit}} \text{ [te-else]}$$

Ces nouveaux connecteurs préservent le type.

Proposition 7.1 *Correction des connecteurs* **then** *et* **else** .

$$\gamma \rhd e_1 \textbf{ then } e_2 : t \quad \Rightarrow \quad \left\{ \begin{array}{l} \textit{pour tout contexte } \mathcal{S} \textit{ vérifiant } \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash e_1 \textbf{ then } e_2] \rightarrow [\mathcal{S}' \vdash e'] \\ \textit{avec } \gamma \rhd e' : t \textit{ et } \Vdash \mathcal{S}' : \gamma \end{array} \right.$$

Preuve : Sans difficultés □

Le typage minimal nous permettra de propager les informations opérationnelles apportées par le système de types, comme par exemple, la non-terminaison de $*(x = x + 1 \textbf{ then } x = x + 1)$, qui se traduit par une erreur lors du contrôle de types.

Proposition 7.2 *Typage minimal des connecteurs* **then** *et* **else** .

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{none}\} \quad \gamma \rhd e_2 : \textbf{Unit}}{\gamma \rhd e_1 \textbf{ then } e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \; [\textit{te-then-min}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_2 : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 \textbf{ then } e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \; [\textit{te-then-min2}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_2 : \textbf{Unit in } \{\textbf{unit}\}}{\gamma \rhd e_1 \textbf{ then } e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \; [\textit{te-then-min3}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd e_1 \textbf{ then } e_2 \preccurlyeq: \textbf{Unit}} \; [\textit{te-then-min4}]$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd e_1 \textbf{ then } e_2 \preccurlyeq: \textbf{Unit}} \; [\textit{te-then-min5}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit} \quad \gamma \rhd e_2 : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 \textbf{ else } e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \; [\textit{te-else-min}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{none}\} \quad \gamma \rhd e_2 : \textbf{Unit in } \{\textbf{none}\}}{\gamma \rhd e_1 \textbf{ else } e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{none}\}} \; [\textit{te-else-min2}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{unit}\} \quad \gamma \rhd e_2 : \textbf{Unit}}{\gamma \rhd e_1 \textbf{ else } e_2 \preccurlyeq: \textbf{Unit in } \{\textbf{unit}\}} \; [\textit{te-else-min3}]$$

$$\frac{\gamma \rhd e_1 : \textbf{Unit in } \{\textbf{none}\} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd e_1 \textbf{ else } e_2 \preccurlyeq: \textbf{Unit}} \; [\textit{te-else-min4}]$$

$$\frac{\gamma \rhd e_1 \preccurlyeq: \textbf{Unit} \quad \gamma \rhd e_2 \preccurlyeq: \textbf{Unit}}{\gamma \rhd e_1 \textbf{ else } e_2 \preccurlyeq: \textbf{Unit}} \; [\textit{te-else-min5}]$$

Preuve : La préservation du type nous permet d' appliquer les hypothèses (numérateurs) à l'expréssion sémantiquement équivalente **if then else** (cf [e-then] et [e-else]). Nous pouvons ensuite utiliser les résultats sur le typage minimal du test (cf le chapitre sur le noyau). □

7.3 Machines abstraites

7.3.1 Syntaxe

Le deuxième groupe de définition porte sur des opérateurs similaires, mais adaptés à la composition de machines abstraites, et non àla composition d'instructions impératives ou règles. Notons l'opérateur **Before** qui correspond à une mise en séquence. L'opérateur $\circledast(e)$ est original : la machine "internalise" l'itération, en ce sens qu'après la composition, l'opérateur d'itération est applicable au corps de la machine abstraite. Si une machine m définit une unique action (en l'occurence, tenter de lire le premier élément d'une liste quelconque, puis l'effacer)

$$\begin{array}{l}\nabla x : [\top], y : \top. \\ \qquad\qquad \textbf{var}\ a : \top. \\ \qquad\qquad \textbf{var}\ v : [\top]. \\ \qquad\qquad \{ \\ \qquad\qquad r : x \sharp [?a] \boxplus ?v \quad \Rightarrow \quad x = v \\ \qquad\qquad \}\end{array}$$

alors $\circledast(m)$ est la machine abstraite qui va parcourir complètement toute liste fournie en paramètres

e	::=	e_1 **Then** e_2	*composer les machines e_1 et e_2 avec la sémantique de* **then**
e	::=	e_1 **Else** e_2	*composer les machines e_1 et e_2 avec la sémantique de* **else**
e	::=	e_1 **Before** e_2	*composer les machines e_1 et e_2 avec la sémantique de* ;
e	::=	$\circledast(e_1)$	*modifier la définition de la machine e_1 avec la sémantique de* $*$

7.3.2 Sémantique

Comme pour $+$, il y a précédence à gauche dans lévaluation.

$$\bigstar \in \{\ \textbf{Then}\ ,\ \textbf{Else}\ ,\ \textbf{Either}\ \}$$

$$\frac{[\mathcal{S} \vdash e_1] \rightarrow [\mathcal{S} \vdash e_1']}{[\mathcal{S} \vdash e_1 \bigstar e_2] \rightarrow [\mathcal{S} \vdash e_1' \bigstar e_2]}\ \text{[e-}\bigstar\text{-L]}$$

$$\frac{[\mathcal{S} \vdash e_2] \rightarrow [\mathcal{S} \vdash e_2']}{[\mathcal{S} \vdash \nabla x : u_1, y : u_2 . m \bigstar e_2] \rightarrow [\mathcal{S} \vdash \nabla x : u_1, y : u_2 . m \bigstar e_2']}\ \text{[e-}\bigstar\text{-R]}$$

A présent, nous décrivons la composition proprement dite. Les deux machines m_1 et m_2 sont transformées en une nouvelle machine m_3 de la manière suivante [e-★-map] : le nom des paramètres d'entrée et sortie de m_3 sont les mêmes que ceux de m_1. Les types associés sont calculés par intersection maximale. L'opérateur est propagé à l'intérieur du corps de m_3, avec comme opérande gauche et droite le corps de m_1 et m_2. Notons toutefois que dans ce dernier, les noms des nouveaux paramètres ont étés substitués.

Ensuite La règle [e-nest-map] montre que la transformation se poursuit dans le corps de m_3, au moyen d'une nouvelle transition notée $\rightarrowtail$, dans le numérateur.

$$\bigstar \in \{\ \textbf{Then}\ ,\ \textbf{Else}\ ,\ \textbf{Before}\ \} \quad \star \in \{\ \textbf{then}\ ,\ \textbf{else}\ \}$$

$$\frac{t \equiv t_1 \otimes t_1' \quad t' \equiv t_2 \otimes t_2' \quad m_2' \equiv m_2[x/x', y/y']}{[\mathcal{S} \vdash (\nabla x : t_1, y : t_2 . m_1)\bigstar(\nabla x' : t_1', y' : t_2' . m_2)] \rightarrow [\mathcal{S} \vdash \nabla x : t, y : t' . (m_1 \bigstar m_2')]} \text{ [e-}\bigstar\text{-map]}$$

$$\frac{[\mathcal{S} \vdash m] \rightarrowtail [\mathcal{S} \vdash m']}{[\mathcal{S} \vdash \nabla x : u, y : u' . m] \rightarrow [\mathcal{S} \vdash \nabla x : u, y : u' . m']} \text{ [e-nest-map]}$$

Nous décrivons à présent cette nouvelle transition qui permet de calculer la composition dans le corps d'une machine.

$$\bigstar \in \{\ \textbf{Then}\ ,\ \textbf{Else}\ ,\ \textbf{Either}\ \} \quad \star \in \{\ \textbf{then}\ ,\ \textbf{else}\ ,\ \textbf{either}\ \}$$

$$\frac{[\mathcal{S} \vdash m] \rightarrowtail [\mathcal{S} \vdash m']}{[\mathcal{S} \vdash \textbf{var}\ x : t_1 . m] \rightarrowtail [\mathcal{S} \vdash \textbf{var}\ x : t_1 . m']} \text{ [e-nest-map2]}$$

$$[\mathcal{S} \vdash (\textbf{var}\ x : t_1 . m_1)\bigstar m_2] \rightarrowtail [\mathcal{S} \vdash \textbf{var}\ x : t_1 . (m_1 \bigstar m_2)] \text{ [e-}\bigstar\text{-map1]}$$

$$[\mathcal{S} \vdash \{\ell_1 : e_1, \cdots, \ell_n : e_n\}\bigstar \textbf{var}\ x : t . m] \rightarrowtail [\mathcal{S} \vdash \textbf{var}\ x : t . (\{\ell_1 : e_1, \cdots, \ell_n : e_n\}\bigstar m)] \text{ [e-}\bigstar\text{-map2]}$$

$$[\mathcal{S} \vdash (e_1 \star e_n)\bigstar \textbf{var}\ x : t . m] \rightarrowtail [\mathcal{S} \vdash \textbf{var}\ x : t . (e_1 \star e_n)\bigstar m] \text{ [e-}\bigstar\text{-}\star\text{]}$$

$$[\mathcal{S} \vdash *(e_1)\bigstar \textbf{var}\ x : t . m] \rightarrowtail [\mathcal{S} \vdash \textbf{var}\ x : t . (*(e_1)\bigstar m)] \text{ [e-}\bigstar\text{-iter]}$$

Le calcul de la composition par itération est plus simple, mais suit le même principe de propager l'opérateur de manière à externaliser les variables locales.

$$[\mathcal{S} \vdash \circledast(\nabla x : t_1, y : t_2 . m)] \rightarrowtail [\mathcal{S} \vdash \nabla x : t, y : t' . \circledast (m)] \text{ [e-iter-map]}$$

$$[\mathcal{S} \vdash \circledast(\textbf{var}\ x : t_1 . m)] \rightarrowtail [\mathcal{S} \vdash \textbf{var}\ x : t_1 . \circledast(m)] \text{ [e-iter-map1]}$$

Enfin, les équations suivantes précisent les étapes terminales du calcul des différentes compositions :

$$[\mathcal{S} \vdash m_1\ \textbf{Then}\ m_2] \rightarrowtail [\mathcal{S} \vdash m_1\ \textbf{then}\ m_2] \text{ [e-then-end]}$$

$$[\mathcal{S} \vdash m_1\ \textbf{Else}\ m_2] \rightarrowtail [\mathcal{S} \vdash m_1\ \textbf{else}\ m_2] \text{ [e-else-end]}$$

$$[\mathcal{S} \vdash m_1\ \textbf{Before}\ m_2] \rightarrowtail [\mathcal{S} \vdash m_1 ; m_2] \text{ [e-before-end]}$$

$$\star \in \{\ \textbf{then}\ ,\ \textbf{else}\ \}$$

$$[\mathcal{S} \vdash \circledast(m_1 \star m_2)] \rightarrowtail [\mathcal{S} \vdash *(m_1 \star m_2)] \quad \text{[e-iter-end]}$$

$$[\mathcal{S} \vdash \circledast(\{\cdots, r : e, \cdots\})] \rightarrowtail [\mathcal{S} \vdash *(\{\cdots, r : e, \cdots\})] \quad \text{[e-iter-end2]}$$

Les formes terminales dérivées par les équations précédentes sont normales pour la relation $\rightarrowtail$, et donc pour $\rightarrow$ (cf [e-nest-map] et [e-nest-map2]).

$$[\mathcal{S} \vdash m_1\ \textbf{then}\ m_2] \rightarrowtail [\mathcal{S} \vdash m_1\ \textbf{then}\ m_2] \quad \text{[e-then-norm]}$$

$$[\mathcal{S} \vdash m_1\ \textbf{else}\ m_2] \rightarrowtail [\mathcal{S} \vdash m_1\ \textbf{else}\ m_2] \quad \text{[e-else-norm]}$$

$$[\mathcal{S} \vdash m_1 ; m_2] \rightarrowtail [\mathcal{S} \vdash m_1 ; m_2] \quad \text{[e-before-norm]}$$

$$\star \in \{\ \textbf{then}\ ,\ \textbf{else}\ , ;\ \}$$

$$[\mathcal{S} \vdash *(m_1 \star m_2)] \rightarrowtail [\mathcal{S} \vdash *(m_1 \star m_2)] \quad \text{[e-iter-norm]}$$

$$[\mathcal{S} \vdash *(\{\cdots, r : e, \cdots\})] \rightarrowtail [\mathcal{S} \vdash *(\{\cdots, r : e, \cdots\})] \quad \text{[e-iter-norm2]}$$

Nous produisons à présent un exemple de dérivation portant sur une *map* qui réécrit une sous-chaîne "bw" en "w". Cette machine réalise une seule opération de réécriture permettant d'implanter le jeux *coffe can game*, présenté en [44]. Plus précisement, la machine proposée en exemple ne réalise qu'une seule substitution, alors que dans les systèmes de réécriture, les substitutions sont réalisées tant qu'un redex est identifiable dans le contexte.

$$\begin{array}{l}\textbf{const}\ m_1 := \nabla x : \textbf{string}, y : \textbf{string}.\\ \quad \textbf{var}\ s_1 : \textbf{string}.\\ \quad \{\\ \quad r_1 : (x \sharp ?s_1 \boxplus \%\text{“}bw'' \boxplus ?s_2) \Rightarrow (x = s_1 +'' w'' + s_2)\\ \quad \}\end{array}$$

Notons de plus, que cette machine retourne toujours **none**, car y n'est jamais modifié. La machine m_2, obtenue par composition, permet d'appliquer la substitution tant que possible :

$$\textbf{const}\ m_2 := \circledast(m_1)$$

La machine m_2 est réduite par la succession d'étapes suivante (l'environnement est volontairement omis) :

$$\begin{array}{rcll}\circledast(m_1) & \rightarrow & \circledast(\nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.\{r_1 : \cdots\}) & \\ & \rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\ \circledast\ (\textbf{var}\ s_1 : \textbf{string}.\{r_1 : \cdots\}) & \text{[e-iter-map]}\\ & \rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.\circledast(\{r_1 : \cdots\}) & \text{[e-nest-map][e-iter-map1]}\\ & \rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.*(\{r_1 : \cdots\}) & \text{[e-iter-end]}\end{array}$$

Cette forme est bien normale, car aucune autre règle ne permet de réécrire le terme différement. Si nous voulons retourner la châine transformée par la réecriture, nous pouvons utiliser la machine m_3 :

$$\textbf{const}\ mm := \nabla x' : \top, y' : \top.\{r : y' = x'\}$$
$$\textbf{const}\ m_3 := m_2\ \textbf{Then}\ mm$$

Cette machine est normalisée par :

$$\begin{array}{lll}
 & m_2\ \textbf{Then}\ mm & \\
\rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.*(\{r_1 : \cdots\})\ \textbf{Then}\ mm & [a] \\
\rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.*(\{r_1 : \cdots\})\ \textbf{Then}\ \nabla x' : \top, y' : \top.\{r : y' = x'\} & [b] \\
\rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.(\textbf{var}\ s_1 : \textbf{string}.*(\{r_1 : \cdots\})\ \textbf{Then}\ \{r : y = x\}) & [c] \\
\rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.*(\{r_1 : \cdots\})\ \textbf{Then}\ \{r : y = x\} & [d] \\
\rightarrow & \nabla x : \textbf{string}, y : \textbf{string}.\textbf{var}\ s_1 : \textbf{string}.*(\{r_1 : \cdots\})\ \textbf{then}\ \{r : y = x\} & [e]
\end{array}$$

[a] [e-then-L][e-var]
[b] [e-then-R][e-var]
[c] [e-nest-map][e-then-map]
[d] [e-nest-map][e-nest-map2][e-iter-map1]
[e] [e-nest-map][e-nest-map2][e-then-end]

Notons que $\textbf{string} \otimes \top \stackrel{def}{=} \textbf{string}$, et que x', y' sont substitués par x, y dans la troisième dérivation.

7.3.3 Typage

La composition par itération ne pose pas de problème. Pour les autres opérateurs, il faut tenir compte du problème évoqué dans notre introduction : après réduction, la nouvelle machine n'est pas forcément bien typée. Les règles [te-Then-map],[te-Else-map] et [te-Before-map] réalisent un contrôle de type après réduction, a partir de la signature transformée. Notons que cette approche interdit la composition "dynamique" : toute machine doit être définie au moment du contrôle de types (sauf pour la composition itérative).

$$\bigstar \in \{\ \textbf{Then}\ ,\ \textbf{Else}\ ,\ \textbf{Before}\ \}$$

$$\frac{\gamma \rhd e : t_1 \Rightarrow t_2}{\gamma \rhd \circledast(e) : t_1 \Rightarrow t_2}\ [\text{te-iter-map}]$$

$$\frac{\begin{array}{c}\gamma \rhd e_1 : t_1 \Rightarrow t_2 \quad \gamma \rhd e_2 : t_1' \Rightarrow t_2' \\ \Vdash \mathcal{S} : \gamma \\ [\mathcal{S} \vdash e_1 \bigstar e_2] \twoheadrightarrow\!\circ\ [\mathcal{S} \vdash v] \\ \gamma \rhd v : (t_1 \otimes t_1') \Rightarrow (t_2 \otimes t_2')\end{array}}{\gamma \rhd e_1 \bigstar e_2 : (t_1 \otimes t_1') \Rightarrow (t_2 \otimes t_2')}\ [\text{te-}\bigstar\text{-map}]$$

7.3.4 Correction

Proposition 7.3 *Correction de la composition.*

Toute composition de machines abstraites bien typée reste bien typée après réduction

Preuve : Directement par la nature même de [te-★-map], qui utilise cette propriété en hypothèse. Les transitions $\longmapsto$ ne sont pas prises en compte, car elle sont complètes vis à vis de la syntaxe du corps d'une machine abstraite, et ne produisent pas d'erreurs. □

7.4 Synthèse

Ce chapitre conclut notre étude théorique en aboutissant sur la composition syntaxique, contribution essentielle de Circus. Trois opérateurs binaires et un opérateur unaire permettent de construire de nouvelles machines abstraites à partir de machines existantes, avec une sémantique pertinente, en ce sens qu'elle correspond bien à leur schéma d'exécution interne (enchainements de règles). Une fois encore, les opérations générales sur les types ($\otimes$ et de manière indirecte $\oplus$) permettent de calculer simplement les signatures associées aux compositions et préservent au mieux le polymorphisme. Si de nombreux autres opérateurs sont imaginables, nous avons proposé un jeu réduit à ce qui semble être l'essentiel. Il est certain à présent qu'une large place doit être laissée à l'expérimentation afin d'identifier les problèmes potentiels et les points fort à développer. Il est clair que ces compositions, associées au système de type statique laisse entrevoir de nombreuses possibilité d'optimisation lors de la compilation.

Troisième partie

Architectures transformationnelles et interactives

Introduction

Dans cette partie, nous nous attachons à illustrer les concepts originaux de *Circus* en montrant comment il peut être utilisé pour réaliser un langage textuel *et* visuel, possédant des noyaux de traitement communs. Préliminairement, les paragraphes suivants complètent la description du langage, d'une part en revenant sur les propriétés compositionnelles des *MAP*, et d'autre part en décrivant les solutions adoptées pour traiter l'analyse lexicale et syntaxique.

Composition de *MAP* L'étude théorique fait apparaître deux sortes de composition : la première est "syntaxique", en ce sens que les opérateurs agissent sur la structure des machines *avant* leur traduction. Par opposition, la seconde est "opérationnelle", puisque des machines traduites sont exécutées selon des modalités différentes (en parallèle avec | ou en séquence avec ;).

L'interêt de la composition syntaxique est triple : diminution de la taille des spécifications, amélioration de la réutilisation et préservation des performances. Spécifier une machine abstraite de taille réduite améliore la lisibilité, facilite les tests et augmente par la suite les possibilités combinatoires. Ce sont précisément ces dernières qui rendent la réutilisation efficace, dans la mesure où les performances ne sont pas pénalisées par le coût de la composition. Ce dernier doit être apprécié en terme de performances d'exécution, mais aussi d'ingéniérie : quelle quantité de code doit on écrire pour valider la composition ? Dans les modèles à objets, chaque méthode surchargée doit être entièrement écrite : l'héritage sans surcharge n'apporte aucune nouvelle propriété dans l'exécution. Il en est tout autrement avec les *MAP* de *Circus*, où une composition simple, sans aucun rajout de code, peut amener de riches changements sémantiques. Cette particularité, qui sera abondamment illustrée dans le reste du développement, présente un intérêt tout à fait général, à rapprocher des réflexions menées récemment sur un nouveau paradigme de programmation, l'*AOP* ("*Aspect Oriented Paradigm*" [**?**]), qui tente d'établir des méthodes de réutilisation plus performantes, plus automatisées, et possédant un grain plus fin que la surcharge propre à la programmation à objets. Les performances sont préservées par des opérateurs de composition syntaxique, car le compilateur possède le contexte d'ensemble pour orienter sa traduction, et donc son optimisation.

L'intérêt, bien connu, de disposer des schémas d'exécution parallèle et séquentielle, est d'enrichir le modèle descriptif des applications. La mise en concurrence des processus de traitement est souvent un facteur d'accroissement des performances. Mais il est également simplificateur, car dans la réalité, des processus n'étant pas en relation de dépendance causale *sont* naturellement parallèles. La mise en séquence est généralement un moyen d'expliciter ce type de dépendances. Dans les architectures des compilateurs et de leurs outils, c'est l'aspect performance qui justifiait l'étude des aspects parallèles (gains quantitatifs en temps de réponse). Avec l'approche que nous développons dans *Circus*, c'est l'interaction avec l'utilisateur qui justifie l'utilisation de schémas d'exécution concurrents (gain qualitatifs). Par exemple, le besoin de conserver une interaction fluide avec l'utilisateur nécessite de lancer en parallèle un traitement complexe, trop long pour être traité dans la continuité de l'action qui l'a déclenché.

Analyse lexicale Seule l'analyse lexicale de texte est considérée, car pour les traitements visuels, les objets graphiques sont considérés comme suffisamment structurés pour être traités directement par une analyse syntaxique. En *Circus*, une construction particulière permet de spécifier des analyseurs lexicaux au moyen d'expressions régulières à la *Lex*. Des tables sont générées, ainsi qu'une machine abstraite réalisant le parcour et la segmentation du texte d'entrée. Cette dernière doit être composée avec d'autre *Map* afin de construire le résultat final (analyse en largeur d'abord) ou d'attacher des actions aux

```
tokenizer tstTok
        {
        alphabet: StdAllSet ;
        tokens:

                Id : '[A-Za-z_][A-Za-z_0-9]*'
                        output 'Identifier'
                        as 'OneId','With_under','Err0_012','__two__'
                        not as '01qwe';

                spaces: '[ \f\n\t]*' ;

                decimal: '[0-9][0-9]*' output 'decimal'
                        as '123','3453453453453','0','00','0123'
                        ;

                Tfloat: '([0-9]*%.[0-9]+)|([0-9]+%.[0-9]*)((E|e)[%+|%-]?[0-9]+)?'
                        output 'Tfloat'
                        as '0.','1.','.01','2.34','1.E+02','1.e+02',
                           '1.34E123','234.00E-19056'
                        not as '12.4E','.','123';
        reserved :

                Id: 'while', 'true', 'false', 'if', 'then', 'else' ;

        operators:

                '+','-','*','/',':=' ;

        delimiters:

                '(',')',';';

        comments:

                ('//','\n'), ('#','\n'), ('/*','*/');

        strings: ('"','"','\\');

        }
```

FIG. 7.1 : l'analyseur lexical de While spécifié en *Circus*

différentes phases de l'analyse (profondeur d'abord, traduction dirigée par l'analyse). La figure 7.1 donne comme exemple l'analyseur lexical de *While* spécifié en *Circus*. Nous ne détaillerons pas l'ensemble des caractéristiques qui ne présentent pas d'interêt scientifique particulier. Les différentes sections ont la signification suivante :

1. "alphabet" : L'alphabet de base sur lequel travaille l'analyseur.
2. "tokens" : introduit les différents lexèmes à l'aide d'un nom logique suivi d'une expression régulière. Optionnellement, la clause "output nom" permet d'exprimer que le lexème doit être conservé avec la catégorie lexicale "nom". Les clauses optionnelles "as" et "not as" illustrent le lexème (souvent, les expressions régulières sont difficiles à lire). De plus elles servent de jeu de test en ligne lors de la vérification.
3. "reserved" : définit les lexèmes réservés dans une catégorie donnée.
4. "operators" : chaines simples
5. "delimiters" : idem
6. "comments" : chaines simples définissant le début et la fin des commentaires (ignorés par l'analyseur)
7. "strings" : idem, avec en plus le caractère d'échappement.

Circus génère un automate déterministe pour chaque catégorie lexicale en utilisant des techniques classiques (cf [4]). Les automates sont encodés au moyen d'un dictionnaire $\{num : \{string : num\}\}$, qui est interprété par la machine abstraite associée pour reconnaitre les différents lexèmes.

Analyse syntaxique Elle est réalisé par une machine abstraite *Circus* qui interprète des tables d'analyse générée à partir d'une grammaire hors-contexte LR(1). La machine de base assure uniquement la reconnaissance des phrases du langage décrit par la grammaire. Des comportements plus sophistiqué sont obtenus par composition avec d'autre machines. Les grammaires sont spécifiées séparément, de manière modulaire, en utilisant une notation proche de *YACC*, dans laquelle chaque non-terminal est associé à une règle et une seule. Ainsi, par exemple, les règles

$$\begin{array}{lcl} A & \rightarrow & B\ \text{';'}\ A \\ A & \rightarrow & B \end{array}$$

doivent s'écrire ("|" signifie "ou")

$$A \quad \rightarrow \quad B\ \text{';'}\ A \quad | \quad B$$

De cette manière, il devient possible de définir des opérations de composition sur les grammaires dont les règles peuvent être désignées sans ambiguité par le non-terminal en partie gauche. En nous inspirant des travaux de [5], nous proposons une relation d'héritage avec surcharge de règles : les constructions *Circus* suivantes,

$$\begin{array}{l} \textbf{grammar} \quad G_1 \quad \{ \\ \langle B \rangle \quad \rightarrow \quad Identifier \quad | \quad num\,; \\ \} \end{array}$$

$$\begin{array}{l} \textbf{grammar} \quad G_2 \quad : G_1 \{ \\ \langle A \rangle \quad \rightarrow \quad \langle B \rangle\ \text{';'}\ \langle A \rangle \quad | \quad \langle B \rangle\,; \\ \} \end{array}$$

montrent comment une grammaire G_2 peut réutiliser une sous-grammaire G_1 (règle définissant $\langle B \rangle$). La grammaire suivante illustre une réutilisation plus complexe : une règle est surchargée par extension, en rajoutant une partie droite.

$$
\begin{array}{lll}
\textbf{grammar} & G_3 & : G_2\{ \\
\langle A \rangle & + \rightarrow & \langle C \rangle \text{ ';' } \langle A \rangle \quad | \quad \langle C \rangle ; \\
\langle C \rangle & \rightarrow & Identifier \text{ '.' } \langle C \rangle \quad | \quad Identifier ; \\
\} &&
\end{array}
$$

L'opérateur $+ \rightarrow$ permet de rendre explicite la modification par extension (sinon, la redéfinition simple d'une règle est considérée comme une erreur). Enfin, la grammaire suivante illustre les possibilités de surcharge par substitution.

$$
\begin{array}{lll}
\textbf{grammar} & G_4 & : G_3\{ \\
\langle C \rangle & - \rightarrow & Identifier \text{ '..' } \langle C \rangle \quad | \quad Identifier ; \\
\} &&
\end{array}
$$

La génération de l'analyseur est indépendante de la grammaire, en ce sens que des grammaires composées sans erreur peuvent révéler des ambiguités (conflits "reduce-reduce" ou "shift-shift") lors de la synthèse des tables. Ces erreurs sont signalées, et doivent être prises en compte pour modifier la grammaire et la rendre LR(1). L'analyse elle-même est éffectuée par une machine interprétant les tables d'analyse $action$ et $goto$, dont le type est,

$$
\begin{array}{lcl}
\textbf{type } lexem & = & \langle cat : \textbf{string}, val : \top \rangle \\
 & & \\
\textbf{type } actionT & = & \{\textbf{num} : \{\textbf{string} : lexem\}\} \\
\textbf{type } gotoT & = & \{\textbf{num} : \{\textbf{num} : \textbf{num}\}\}
\end{array}
$$

Le code circus de l'analyseur est une composition de plusieurs machines abstraites :

$$
\begin{array}{lcl}
\textbf{const } SHIFT & := & 0 \\
\textbf{const } REDUCE & := & 1 \\
\textbf{type } itemS & = & \langle class : \textbf{num in} \{SHIFT\}, newState : \textbf{num} \rangle \\
\textbf{type } itemR & = & \langle class : \textbf{num in} \{REDUCE\}, offset : \textbf{num}, rule : \textbf{num} \rangle \\
\textbf{type } item & = & itemS \otimes itemR
\end{array}
$$

```
const UNKNOWN_LEX   :  num = 1;
const UNEXP_EOF     :  num = 2;
from G4              import action;
from G4              import goto;

const scanner       :=  ∇x:⟨inp:[⊤]⟩, y:⟨lex:⊤⟩.
                        var v:⊤.
                        var lex:⊤.
                        {
                          inp:x.inp ⇵ [?lex]⊞?v  ⇒ (x.inp = v; y.lex = lex)
                        }

const Fetch         :=  ∇x:⟨stack:[num]⟩, out:⟨lex:lexem, it:item, error:[⟨string, lexem⟩]⟩.
                        var s:num.
                        var n:num.
                        var d:{string:item}.
                        var it:item.
                        {
                          tst:
                          if out.error == [] then
                            x.stack ⇵ ?s⊞[?n]
                              ⇒ action ⇵ ?⊞{n = ?d}
                                  {
                                  t1:d ⇵ ?⊞{out.lex.cat = ?it} ⇒ out.it = it,
                                  error: out.error = out.error + [⟨'unexpected lexem', out.lex⟩]
                                  }
                          else
                            none
                        }

  const Shift       :=  ∇x:⟨stack:[num]⟩, output:⟨it:item⟩.
                        var ns:num.
                        {
                          shift:output.it ⇵ ⟨class = %SHIFT, newState = ?ns⟩
                              ⇒ x.stack = x.stack + [ns]
                        }

  const Reduce      :=  ∇x:⟨stack:[num], inp:[⊤]⟩, y:⟨lex:lexem, it:item⟩.
                        var off:num.
                        var r:num.
                        var ns:num.
                        var st:[num].
                        {
                          red:y.it ⇵ ⟨class = %REDUCE, offset = ?off, rule = ?r⟩
                              ⇒
                              *((off < 0) → (x.stack ⇵ ?st⊞[?] ⇒ x.stack = st); off = off + 1);
                              x.stack ⇵ ?⊞[?ns]
                                ⇒ goto ⇵ ?⊞{ns = ?⊞{r = ?ns}}
                                ⇒ x.stack = x.stack + [ns]; x.inp = x.inp + [y.lex]
                        }

const parserM1 := scanner Then Fetch Then (Shift Else Reduce)
const parserM := ⊛(parserM1)
```

Cette machine assure la reconnaissance uniquement, mais ne construit pas d'arbre syntaxique. Elle utilise la technique de reconnaissance LR(1), décrite en [4], mais avec une gestion de pile réduite à son minimum (ici, les états de l'analyseur). Pour construire un reconnaisseur plus évolué, par exemple capable de bâtir un arbre syntaxique, il faut composer $parserM$ avec une machine telle que $parserR$ (qui crée ici un arbre sous forme d'emboîtement de listes) :

$$
\begin{array}{lll}
\textbf{const}\ parserRS & := & \nabla x{:}\top, y{:}\langle lex : lexem, bstack : [\top]\rangle. \\
 & & \textbf{var}\ ns : \textbf{num}. \\
 & & \{ \\
 & & \quad bshift : y.stack = y.stack + [lex] \\
 & & \} \\
 & & \\
\textbf{const}\ parserRR & := & \nabla x{:}\top, y{:}\langle it : item, bstack : [\top]\rangle. \\
 & & \textbf{var}\ ns : \textbf{num}. \\
 & & \{ \\
 & & \quad bred : y.it \ \sharp\ \langle offset = ?off\rangle \\
 & & \qquad \Rightarrow \\
 & & \qquad\quad off = 0; st = [\,]; \\
 & & \qquad\quad *((off < 0) \Rightarrow ((y.bstack \ \sharp\ ?\boxplus[?v] \Rightarrow st = st + [v]); off = off + 1)); \\
 & & \qquad\quad y.bstack = y.bstack + [st]; \\
 & & \}
\end{array}
$$

Le parser complet est crée par une nouvelle composition

$$\textbf{const}\ parserR := \circledast(scanner\ \textbf{Then}\ Fetch\ \textbf{Then}\ ((Shift\ \textbf{Before}\ ParserRS)\ \textbf{Else}\ (Reduce\ \textbf{Before}\ ParserRR)))$$

Sa signature est

$$\langle inp : [\top], stack : [\textbf{num}]\rangle \Longrightarrow \langle lex : lexem, it : item, bstack : [\top], error : [\langle \textbf{string}, lexem\rangle]\rangle$$

et nous donnons un exemple d'utilisation

$$
\begin{array}{l}
\textbf{var}\ str : [lexem] = Lex(). \\
\quad A = parserR(\langle inp = str, stack = [0]\rangle); \\
\quad A.error \ \sharp\ \%[] \Rightarrow b = A.bstack; \cdots \\
\quad \textbf{else} \\
\qquad \textbf{var}\ s : \textbf{string} = ''.\textbf{var}\ l : lexem = \textbf{none}. \\
\qquad\quad A.error \ \sharp\ [\langle ?s, ?l\rangle] \Rightarrow \textbf{put}\ s + Str(l) : \textbf{string} \\
\quad ;
\end{array}
$$

On peut naturellement imaginer d'autres compositions pour attacher des actions aux différentes phases de l'analyse : par exemple, construire un arbre dit "threaded tree" pour l'analyse incrémentale [71], ou implanter des traitements de récupération d'erreurs, telles que la resynchronisation sur le flot d'entrée par recherche de séparateurs [78, 4].

Machines visuelles

8.1 Introduction : représentation et interaction

Dans ce chapitre, nous abordons deux problèmes distincts en tentant de les traiter dans le même cadre : la représentation visuelle d'informations et la gestion des interactions. Du point de vue de l'utilisateur humain, ces deux aspects sont fortement liés, puisque ce dernier agit sur l'environnement qu'il perçoit, et que la représentation est modifiée par les actions de l'utilisateur. C'est précisément l'essence de l'interactivité. Pourtant, il est important de différencier les problèmes de représentation et d'interaction pour plusieurs raisons :

1. La structure spatiale des représentations peut être traitée indépendamment du temps.
   On peut effectivement analyser la syntaxe et la sémantique d'une phrase visuelle en ignorant totalement les étapes et actions qui ont autorisé sa réalisation. En ce sens c'est une simplification, puisqu'une dimension est ignorée. Il est possible alors d'adopter une stratégie similaire aux traitements textuels, en distinguant une couche lexicale et une couche syntaxique. Le première offre une simplification en autorisant la catégorisation des signes, qui permet à la seconde de se concentrer sur les structures plus importantes [1].
2. La structure des actions de l'utilisateur est linéaire dans le temps.
   Celles-ci peuvent donc être considérées comme unidimensionnelles et ordonnées, donc traitables par les techniques classiques d'analyse syntaxique, à condition toutefois de tenir compte de l'irreversibilité du temps. Concrètement, cela se traduit par un traitement particulier des erreurs syntaxiques, puisque le recouvrement ne peut être obtenu qu'en autorisant de nouveaux circuits d'actions correctives. Des grammaires d'action (cf *UAN*, User Action Notation [58]) sont utilisées pour modéliser le comportement de l'utilisateur. Nous proposons d'utiliser des grammaires LR(1), associées à un analyseur spécifique, pour traiter formellement, et concrètement, les interactions de l'utilisateur.

Le modèle que nous adoptons repose sur une machine abstraite spécifique, dite visuelle, qui transcrit des structures de données et des instructions graphiques de manière à gérer un espace de représentation (perçu par l'utilisateur). De plus cette machine interprète les actions de l'utilisateur, de manière à produire des évènements simples, reliés au contexte visuel, pouvant être assimilés à des lexèmes d'interaction. La structure temporelle de ces lexèmes est analysée au moyen d'une autre machine qui interprète les tables d'analyse LR(1). L'interactivité est rendue possible par l'envoi de code graphique à la machine visuelle, en fonction de l'analyse des actions. Le lecteur pourra trouver une présentation générale de cette approche dans [118], produite en annexe de ce document. Une étude plus approfondie d'un système de type visuel est proposée dans [117], ainsi qu'une formalisation d'inspiration géométrique. Dans cette dernière étude, nous proposons une notion de forme qui généralise les éléments "point", "ligne", "polygones" et "polyèdres" dans un espace euclidien à trois dimensions.

8.2 Types visuels

Nous présentons à présent un système de type visuel, conçu pour être générique (c'est à dire suffisamment expressif pour modéliser un grand nombre de représentations visuelles). L'idée fondamentale est d'obtenir du polymorphisme visuel, c'est à dire une simplification du système de signes basée sur des propriétés communes des "lexèmes" visuels. Définir un centre géométrique unique, pour tous les signes, apporte par exemple une grande simplification : quelque soit l'objet considéré et son niveau de

[1] Toutefois, le temps psychologique lié à l'activité perceptuelle et cognitive ne peut pas être ignoré dans les processus cognitifs [112, 100]

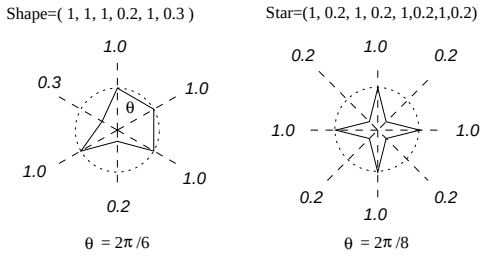


FIG. 8.1 : Deux enveloppes (normalisées) et leurs représentations dans le cercle unitaire

complexité, il est possible de lui appliquer les mêmes opérations géométriques. Nous généralisons cette idée au moyen de la notion d'attributs perceptuels, de glyphes élémentaires et composites.

8.2.1 Attributs perceptuels

Du point de vue de la perception, les attributs de position, taille, forme, couleur et orientations sont "orthogonaux", en ce sens qu'ils peuvent varier indépendamment les uns des autres [15, 16, 17]. Si l'on considère l'espace de représentation comme un espace vectoriel à plusieurs dimensions, ces attributs forment une famille génératrice. Bien entendu, cette métaphore mathématique est limitée, car les études en psychologie cognitive montrent que ces grandeurs ne sont pas linéaires. Toutefois, ce parallèle permet quand même de développer une approche très pure du typage visuel. Ces attributs seront donc transcrits en types *Circus*, en faisant l'hypothèse que leur interprétation par une machine visuelle leur donnera une sémantique perceptuelle. C'est l'essence même de la transposition de niveaux sémantiques, axiome fondamental omniprésent dans le traitement des langages (comme évoqué dans l'état de l'art), qui trouve ici un nouvel éclairage [1]. Les types suivants définissent les attributs visuels :

type A_color	=	$\langle chr : \textbf{num}, lum : \textbf{num}, sat : \textbf{num}\rangle$;	// réels normalisés
type A_size	=	**num**;	// réel positif
type A_orient	=	**num**;	// réel normalisé $\in [-1, 1]$
type A_shape	=	$\langle offset : A\_orient, norm : num, xi : [\textbf{num}]\rangle$;	
type A_pos	=	$\langle x : \textbf{num}, y : \textbf{num}\rangle$;	

Interprétation des attributs visuels

La représentation des couleurs proposée (*chrominance, luminance, saturation*) tient compte des particularités de la perception physiologique ([13]) : l'espace *HSV* (Hue, Saturation, Value) offre la possibilité de variations indépendantes des composantes colorimétriques. Ce n'est pas le cas avec l'espace *RGB*, où par exemple, une variation de la composante verte entraîne également une variation de l'intensité lumineuse. Les deux espaces sont aisément mis en isomorphisme à l'aide d'une matrice 3x3 à coefficients réels. L'attribut de taille est toujours positif. Géométriquement, lorsqu'il est combiné avec l'attribut de forme dans un glyphe, la taille correspond au rayon du cercle minimal contenant l'objet graphique. L'orientation est l'angle du vecteur caractéristique d'un glyphe avec la verticale du système de représentation $\mathcal{SR}$. Le vecteur caractéristique d'un glyphe est un vecteur de norme 1 associé à l'objet graphique de façon à pouvoir définir son orientation dans $\mathcal{SR}$. L'interprétation des formes est plus délicate, et réutilise partiellement nos travaux précédents [117]. Dans les espaces à deux dimensions auxquels nous nous intéressons, ces dernières sont constituées de trois éléments :

[1] Ici, il serait plus approprié de parler de transposition de referentiel sémantique, où les rapports entre sémiotique et sémantique refont à nouveau surface

1. enveloppe ξ_n. C'est une liste ordonnée de valeurs numériques $[v_1, \ldots, v_n]$ réelles. Chaque v_i représente le module d'un vecteur $\overrightarrow{OP_i}$. O est le centre géométrique d'un cercle de rayon unitaire sur lequel chaque point P_i peut être projeté. L'angle θ_n entre les vecteurs $\overrightarrow{OP_{i-1}}$ et $\overrightarrow{OP_i}$ est déterminé par le nombre de points de ξ_n :
$$\theta_n = 2\pi/n$$
Ce qui signifie simplement que les projections des points de l'enveloppe sont uniformément réparties sur le cercle unitaire (c.f figure 8.1).
2. angle de décalage. Il définit une rotation du cercle unitaire par rapport à la verticale de $\mathcal{SR}$. Ainsi, deux formes ayant même ξ_n et un angle de décalage différent sont condidérées comme distinctes, puisque pour un même attribut d'orientation elles sont perçues différemment (par exemple, un triangle isocèle avec une pointe en haut ou une pointe en bas).
3. coefficient normalisateur $C_{\mathcal{N}}$. Il permet de normaliser l'enveloppe en la ramenant à l'intérieur du cercle unitaire. Chaque module doit donc être multiplié par $C_{\mathcal{N}}$ qui est tel que :
$$C_{\mathcal{N}} = 1/max(v_i) \quad , \quad v_i \in \xi_n$$

La structure proposée permet de rendre indépendants les attributs d'orientation, de forme et de taille. De plus, elle autorise l'utilisation d'opérateurs de composition de formes, ce qui est intéressant car la définition de l'aspect des objets visuels est souvent difficile.

8.2.2 Glyphes élémentaires

Tels quels, les attributs perceptuels ne sont pas visualisables (comment représenter une forme sans taille !). Ils sont incorporés à des entités visuelles appelées *glyphes*, qui constituent les signes élémentaires de notre système de représentation. Ceux-ci sont construits de la manière suivante :

$$\begin{array}{lcl}
\textbf{type } Pos & = & \langle pos : A_pos \rangle; \\
\textbf{type } Glyph & = & Pos \otimes \langle orient : A_orient \rangle; \\
\textbf{type } S_Glyph & = & Glyph \otimes \langle size : A_size \rangle; \\
\textbf{type } C_Glyph & = & Glyph \otimes \langle color : A_color \rangle; \\
\textbf{type } SC_Glyph & = & C_Glyph \otimes S_Glyph; \\
\textbf{type } G_Point & = & SC_Glyph \otimes \langle shape : S_point \rangle; \\
\textbf{type } G_Line & = & SC_Glyph \otimes \langle shape : S_line \rangle; \\
\textbf{type } G_Polygon & = & SC_Glyph \otimes \langle shape : A_shape \rangle; \\
\textbf{type } Text & = & SC_Glyph \otimes \langle text : \textbf{string}, font : \textbf{string} \rangle; \\
\textbf{type } Circle & = & SC_Glyph \otimes \langle internal : \textbf{string in} \{'circle'\} \rangle;
\end{array}$$

L'algorithme suivant permet de transposer un glyphe g tel que
$$g = \langle pos = \langle x = a_x, y = a_y \rangle, size = s, orient = \alpha, shape = \langle norm = c_{\mathcal{N}}, offset = o, xi = [v_i] \rangle \rangle$$
en un polygone dont les sommets sont en coordonnées cartésiennes dans le repère d'écran [1]

Définition 8.1 *Algorithme de transposition des glyphes en polygones.*
1-Normaliser ξ_n à l'aide du coefficient $c_{\mathcal{N}}$.
$$\forall i \in [1 \cdots n] \quad , \quad N_i = v_i \times c_{\mathcal{N}}$$
2-Calculer la génératrice V^n (C_x et C_y prennent les valeurs -1 ou 1 en fonction du repère d'écran) :
$$\forall i \in [1 \cdots n] \quad , \quad \left\{ \begin{array}{lcl} \delta & = & (\theta_n \times i) + \alpha + o + \pi/2 \\ V_i.x & = & C_x \times N_i \times sin(\delta) \\ V_i.y & = & C_y \times N_i \times cos(\delta) \end{array} \right.$$
3-Calculer le polygone final P^n par translation de pos :

[1] C'est le moyen usuel pour tracer des polygones dans la plupart des systèmes graphiques

$$\forall i \in [1 \cdots n] \quad , \quad \left\{ \begin{array}{lcl} P_i.x & = & V_i.x + pos.x \\ P_i.y & = & V_i.y + pos.y \end{array} \right.$$

Notons que les différentes phases du calcul peuvent être realisées séparément afin d'optimiser les temps de traitement (les phases 1-2-3 sont invoquées lors d'un changement de forme, les phases 2-3 lors d'un changement d'orientation, et la phase 3 lors d'un changement de position). Les définitions *Circus* suivantes sont des exemples de quelques attributs de base :

type $T_0 = num$ **in** $\{0\}$;
type $T_1 = num$ **in** $\{1\}$;
type $S\_line = A\_shape$ **in** $\{\langle offset : T_0, norm : T_1, xi : [\textbf{num}]$ **in** $\{[1,1]\}\rangle\}$;
type $S\_point = A\_shape$ **in** $\{\langle offset : T_0, norm : T_1, xi : [\textbf{num}]$ **in** $\{[1]\}\rangle\}$;
// quelques polygones
// triangle isocèle
const $S\_ISOT : A\_shape = \langle offset = 0, norm = 1, xi = [1,1,1]\rangle$;
// idem, pointe en bas
const $S\_ISOT\_R : A\_shape = \langle offset = \pi, norm = 1, xi = [1,1,1]\rangle$;
const $S\_SQUARE : A\_shape = \langle offset = 0, norm = 1, xi = [1,1,1,1]\rangle$;
const $S\_LOSANGE : A\_shape = \langle offset = \pi/2, norm = 1, xi = [1,1,1,1]\rangle$;
// étoile 4 branches
const $S\_STAR\_4 : A\_shape = \langle offset = 0, norm = 1, xi = [1, 0.5, 1, 0.5, 1, 0.5, 1, 0.5]\rangle$;

8.2.3 Glyphes composites

C'est un agrégat d'autres glyphes, composites ou non.

$$\textbf{type}\ CompoundGlyph \quad = \quad S\_Glyph \otimes \langle sub : \{S_Glyph\}\rangle;$$

La machine visuelle maintient cette arborescence en cohérence par rapport à la position, l'orientation et la dimension. Les positions des glyphes contenus dans le membre sub sont interprétées comme relatives à la position du glyphe composite. Si la machine visuelle applique un changement d'orientation, de position, ou de taille à un glyphe composite, l'ensemble des "sous-glyphes" est transformé (rotation par rapport à la position centrale, translation ou homothétie). L'idée est donc de considérer un composite comme un glyphe unique, monolithique. On retrouve cette approche constructive des unités lexicales visuelles dans [115]. Toutefois, le polymorphisme "géométrique" inhérent à notre système de glyphes est ici mis à profit.

8.3 Syntaxe de l'interaction

Les événements de base Ils sont constitués des actions souris et des actions clavier. Le déplacement du pointeur à l'intérieur d'un glyphe de type t provoque l'émission d'un événement de type $+t$. Sa sortie provoque un événement complémentaire de type $-t$. De plus la pression du bouton n de la souris provoque un événement $press\_n$, et son relachement $release\_n$. si l'utilisateur presse une clef du clavier, l'événement émis sera $keypressed$ et $keyrelease$ sera associé à son relâchement. Il y a donc une symétrie, en ce sens que chaque événement est associé à son dual au cours d'une série d'actions quelconque entreprise par un utilisateur. Cette particularité sera utilisée pour le traitement automatique des erreurs d'interaction, présenté plus loin. Bien que ce jeu de primitives soit très réduit, il permet de modéliser des combinaisons complexes au moyen de la grammaire d'actions.

Les lexèmes d'interaction Les événements "basiques" doivent être reliés aux glyphes gérés par la machine visuelle, et perçus par l'utilisateur, afin de devenir des "lexèmes" d'interaction. Pour les événements d'entré et sortie, il est nécessaire de déterminer l'instance de glyphe concernée, ainsi que sa catégorie

const $pos1$: $Pos = \langle x : 50, y : 50 \rangle$;
const $pos2$: $Pos = \langle x : 65, y : 65 \rangle$;
const $gseq$: $[GOp]$ = [PUSH, $GStar$, PUSH, 'star', CREATE_LEX]+
[PUSH, $GSquare$, PUSH, 'square', CREATE_LEX]+
[PUSH, 'star', CREATE, PUSH, $pos1$, MOVE_TO]+
[PUSH, 'square', CREATE, PUSH, $pos2$, MOVE_TO];

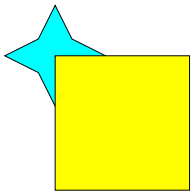


FIG. 8.2 : Une séquence d'instruction graphique $gseq$, et son résultat dans l'espace de travail

POINTEUR	ÉVENEMENT	LEXÈMES
entre dans l'étoile, puis ressort	'+star' '-star'	$\langle class =$ '+star', $val = gid1\rangle$ $\langle class =$ '-star', $val = gid1\rangle$
entre dans le carré, puis ressort	'+square' '-square'	$\langle class =$ '+star', $val = gid2\rangle$ $\langle class =$ '-square', $val = gid2\rangle$
entre dans l'étoile, dans le carré puis sort de l'étoile et du carré	'+star' '+square' '-star' '-square'	$\langle class =$ '+star', $val = gid1\rangle$ $\langle class =$ '+square', $val = gid2\rangle$ $\langle class =$ '-star', $val = gid1\rangle$ $\langle class =$ '-square', $val = gid2\rangle$
entre dans le carré, dans l'étoile puis sort du carré et de l'étoile	'+square' '+star' '-square' '-star'	$\langle class =$ '+square', $val = gid2\rangle$ $\langle class =$ '+star', $val = gid1\rangle$ $\langle class =$ '-square', $val = gid2\rangle$ $\langle class =$ '-star', $val = gid1\rangle$

FIG. 8.3 : Une séquence d'actions, les évenements correspondants et les lexèmes générés

lexicale (ou type visuel). Si par exemple, le code suivant permet la définition de types visuels $GStar$ et $GSquare$:

type $GStar = Glyph \otimes \langle size :$ **num in** $\{10\}, color : A\_color$ **in** $\{$BLUE$\}, shape : A_shape$ **in** $\{S\_STAR\_4\}\rangle$;
type $GSquare = Glyph \otimes \langle size :$ **num in** $\{20\}, color : A\_color$ **in** $\{$YELLOW$\}, shape : A_shape$ **in** $\{S\_SQUARE\}\rangle$;

alors, l'exécution de la séquence de code de la figure 8.2 par la machine visuelle permet de construire un espace de travail dans lequel les actions décrites dans la figure 8.3 sont possibles. La colonne "lexèmes" montre les données *Circus* déposées par la machine visuelle dans la mémoire globale. Le champs val contient la référence de l'objet graphique (instance de glyphe) concerné par l'événement. Ces données d'interactions sont écrites en mémoire au moyen du type suivant :

type $VLex = \langle typ :$ **string in**$\{$'vlex'$\}, val = [lexem]\rangle$;

où val représente la liste des lexèmes d'interaction générés par la machine abstraite visuelle. Le producteur de ce flot d'événements gére le tampon val, au moyen des primitives get et put. Si une première tentative de lecture avec get échoue, c'est que le tampon a été consommé. Dans ce cas il suffit d'écrire dans la mémoire de coordination une nouvelle structure $VLex$. Sinon, il faut concaténer les nouveaux évenements aux anciens et remettre le tout dans la mémoire. Le consommateur utilise simplement $bget$

pour la lecture destructive bloquante. Ces deux machines sont spécifiées par

$$
\begin{array}{lcl}
\textbf{const}\ Producer & := & \nabla x : [lexem], y : \top. \\
 & & \textbf{var}\ v : [\top]. \\
 & & \{ \\
 & & \quad tryit : \textbf{get}\ \langle typ = \%'vlex', val = ?v \rangle \\
 & & \qquad \Rightarrow \textbf{put}\ \langle typ =' vlex', val = v + x \rangle : Vlex \\
 & & \quad putit : \textbf{put}\ \langle typ =' vlex', val = lex \rangle : Vlex \\
 & & \} \\
 & & \\
\textbf{const}\ Consumer & := & \nabla x : \top, y : \top. \\
 & & \textbf{var}\ v : [\top]. \\
 & & \{ \\
 & & \quad get : \textbf{bget}\ \langle typ = \%'vlex', val = ?v \rangle \Rightarrow x = v \\
 & & \}
\end{array}
$$

Reconnaissance d'actions complexes et réactions graphiques L'enchaînement d'actions complexes est spécifiable par une grammaire LR(1) dont les éléments terminaux sont les événements d'interaction. Ainsi, la grammaire suivante capture les interactions décrites en figure 8.3.

$$
\begin{array}{lcl}
All & \rightarrow & St \mid Sq \mid StSq \mid SqSt; \\
St & \rightarrow & \text{'+star' '-star'} \\
Sq & \rightarrow & \text{'+square' '-square'} \\
StSq & \rightarrow & \text{'+star' '+square' '-star' '-square'} \\
SqSt & \rightarrow & \text{'+square' '+star' '-square' '-star'}
\end{array}
$$

Toutefois, nous désirons réaliser des langages visuels moins statiques. Par exemple, pouvoir modifier la couleur d'un glyphe lorsque le pointeur de souris entre dans ce dernier. La méthode habituelle avec les reconnaisseurs "bottom-up" est d'associer des actions aux phases de réduction des règles. Dans notre cas, ce n'est pas suffisant, car le comportement prédictif [1] de l'analyse LR(1) ou LALR(1) n'est pas compatible avec les contraintes de l'interaction : il faut effectivement attendre l'événement suivant pour réduire la règle, ce qui va à l'encontre de la réactivité attendue. Nous proposons donc d'associer les réactions graphiques aux phases de décalage. En utilisant la composition de machines abstraites (opérateur $\odot$), il devient possible de spécifier un reconnaisseur travaillant sur la pile d'analyse pour envoyer des instructions graphiques via la mémoire globale. La machine suivante prépare les actions graphiques capables de modifier la couleur de l'étoile (cf exemple fig. 8.2) en noir quand le pointeur entre dans la forme, et de restaurer la couleur d'origine lorsqu'il en ressort.

$$
\begin{array}{lcl}
\textbf{type}\ VAMop & = & \langle typ : \textbf{string} \in \{'vop'\}, seq = [\top] \rangle \\
 & & \\
\textbf{const}\ setColor & := & \lambda x : \top.[\textbf{PUSH}, x, \textbf{PUSH}, \textbf{BLACK}, \textbf{SET-COLOR}] \\
\textbf{const}\ resetColor & := & \lambda x : \top.[\textbf{PUSH}, x, \textbf{DUP}, \textbf{GET-TYPE}, \textbf{GET-COLOR}, \textbf{SET-COLOR}] \\
 & & \\
\textbf{const}\ ParserI & := & \nabla x : \top, y : \langle bstack : [\top] \rangle \\
 & & \textbf{var}\ id : \textbf{num}. \\
 & & \{ \\
 & & a1 : y.bstack \not\equiv\ ? \boxplus [\langle cat = \%' + star', val = ?id \rangle] \\
 & & \quad \Rightarrow \\
 & & \qquad \textbf{var}\ v : VAMop = \langle typ =' vop', seq = setColor(id) \rangle. \\
 & & \qquad \textbf{put}\ v : VAMop \\
 & & a2 : y.bstack \not\equiv\ ? \boxplus [\langle cat = \%' - star', val = ?id \rangle] \\
 & & \quad \Rightarrow \\
 & & \qquad \textbf{var}\ v : VAMop = \langle typ =' vop', seq = resetColor(id) \rangle. \\
 & & \qquad \textbf{put}\ v : VAMop \\
 & & \}
\end{array}
$$

[1] en ce sens que c'est la lecture du lexème $n+1$ qui determine le comportement de l'analyseur quant au lexème n

Gestion des erreurs Comme évoqué précédemment, elle doit tenir compte de l'irréversibilité du temps : il n'est pas possible de supprimer les actions passées qui ont modifié le cours de l'analyse. Ainsi, l'idée que nous avons mis en œuvre est d'utiliser une pile spéciale, (pile d'erreur) qui mémorise la succession d'événements postérieurs à l'erreur d'analyse. Lorsque l'analyseur détecte une erreur, il dépose le lexème en faute dans la pile d'erreur, puis commute dans un mode spécial. Chaque nouveau lexème est comparé avec le sommet de la pile d'erreur. Lorsqu'ils sont complémentaires, la pile est décrémentée. L'analyseur commute en mode normal lorsque la pile d'erreur est vide (on considère que toutes les contre-actions ont étés réalisées). Cette stratégie utilise la symétrie des événements de base pour offrir une grande souplesse à l'utilisateur : si ce dernier éxécute une action inattendue (par exemple appuyer sur un bouton de la souris), son erreur est signalée (pointeur de souris avec une forme spéciale), et il doit faire la contre-action corrective (relacher le bouton), ce qui est très naturel. En cas d'erreurs multiples, les contre-actions doivent être réalisées dans l'ordre inverse, ce qui peut devenir contraignant. Une variante envisageable serait d'utiliser un ensemble au lieu d'une pile, insensible à l'ordre. Un autre aspect intéressant de cette gestion d'erreurs est lié à une particularité de l'analyse LR(1) : toutes les erreurs syntaxiques sont détectées lors du décalage, et contrairement à l'analyse LALR(1), aucune réduction erronée ne peut survenir. Cette propriété protège les opérations sémantiques profondes des "erreurs de surface" et de ce fait simplifie beaucoup la gestion des erreurs lorsque l'analyse est fortement couplée aux traitements sémantiques. La machine suivante, variante de la machine de base $parserM$, implémente ce comportement

$$
\begin{array}{lll}
\textbf{const}\ ErrManager & := & \nabla x : \top, y : \langle error : [\langle \textbf{string}, lexem \rangle], recover : [], lex : lexem \rangle \\
 & & \textbf{var}\ id : lexem. \\
 & & \textbf{var}\ s : \textbf{string}. \\
 & & \textbf{var}\ l : [\textbf{string}]. \\
 & & \{ \\
 & & err : y.error\ \sharp\ [\langle \%'\text{unexpected lexem}', val = ?id \rangle] \\
 & & \quad \Rightarrow \\
 & & \quad\quad y.recover\ \sharp\ ?l \boxplus [?s] \Rightarrow \\
 & & \quad\quad\quad \text{if } (s == id.cat) \text{ then} \\
 & & \quad\quad\quad\quad y.recover = l; \\
 & & \quad\quad\quad\quad l\ \sharp\ \%[\,] \Rightarrow y.error = [\,] \\
 & & \quad\quad\quad \text{else} \\
 & & \quad\quad\quad\quad y.recover = l + [s] + Reverse(id.cat) \\
 & & \}
\end{array}
$$

La machine Reverse calcule l'évenement symétrique qui permettra de recouvrer la dernière erreur : soit il s'agit d'un évenement souris, soit une pression sur les boutons ou le clavier.

$$
\begin{array}{lll}
\textbf{const}\ Reverse & := & \nabla x : \textbf{string}, y : \textbf{string}. \\
 & & \textbf{var}\ s : \textbf{string}. \\
 & & \{ \\
 & & r1 : x\ \sharp\ \%'+' \boxplus ?s \Rightarrow y =' -' + s, \\
 & & r2 : x\ \sharp\ \%'Press' \boxplus ?s \Rightarrow y =' Release' + s, \\
 & & r3 : x\ \sharp\ \%'KeyPress' \Rightarrow y =' KeyRelease' \\
 & & \}
\end{array}
$$

Enfin, voici la nouvelle composition qui permet d'intégrer la gestion d'erreur à l'analyse des actions utilisateur

$$\textbf{const}\ ParserIE := \circledast(Consumer\ \textbf{Then}\ \circledast(scanner\ \textbf{Then}\ (Fetch\ \textbf{Then}\ ErrManager\ \textbf{Then}\ (Shift\ \textbf{Else}\ Reduce))))$$

Notons qu'il serait nécessaire d'enrichir encore la composition pour detecter l'éventuelle terminaison du processus d'analyse (par exemple, une règle en tête pour tester la présence d'un jeton dans la mémoire de coordination).

8.4 Syntaxe visuelle

Nous avons opté pour un formalisme à base de grammaire *PLR*, décrit en [38, 114], qui possède l'avantage d'être une extension des grammaires LR (meilleure unité des concepts proposés aux concepteurs de langages), d'être efficace dans l'implémentation et qui semble suffisamment expressif (cf [37]). La construction des tables est similaire à la technique "classique", exceptée une table supplémentaire : lors de l'interprétation, chaque phase de décalage doit être précédée de l'évaluation d'un prédicat relationnel (relation spatiale associée aux éléments contenus dans la pile des lexèmes). Ceci implique donc l'existence d'une table contenant les prédicats devant être appliqués aux éléments contenus dans la pile (elle est appelée table "next" dans [35, 38, 36]). La construction de cette table peut générer des conflits, lorsqu'un prédicat relationnel est déjà implanté avec une valeur différente. Ainsi une grammaire est *PLR* lorsque aucun conflit de décalage, reduction et de prédicat n'apparait lors du calcul des tables. Outre l'évaluation des prédicats, il est d'autre part nécessaire de réaliser une action lors de la réduction : l'instanciation des attributs du non-terminal gauche, en fonction des attributs des membres droits. Cette phase est indispensable pour propager les informations spatiales au cours de l'analyse. Enfin, comme pour les syntaxes *LR*, l'analyseur travaille sur un ensemble de lexèmes visuels, et doit connaître l'élément de départ afin de lancer l'analyse.

Spécification des grammaires visuelles Nous décrivons rapidement les extensions syntaxiques que nous devons apporter aux grammaires LR déja présentées, afin de rester homogène, et de disposer des mêmes facilités de composition.

1. *Nommage des termes.* Il est indispensable de pouvoir distinguer les différents termes pour pouvoir spécifier les relations spatiales et la synthèse des attributs. Dans *YACC*, il est possible d'utiliser le nombre donnant la position du terme dans la règle. Cette méthode est peu lisible, et supporte mal les évolutions. Nous préférons la solution qui consiste à nommer explicitement les termes :

   Rgrammar G_5 {
   terms : $square : \langle cat : \textbf{string in}\{'square'\}, val : SC\_Glyph \otimes \langle shape : A\_Shape\textbf{in } \{S\_SQUARE\}\rangle\rangle$;
   $line : \langle cat : \textbf{string in}\{'line'\}, val : G\_Line\rangle$;
   nterms : $A : \langle cat : \textbf{string in}\{'A'\}, val : Pos\rangle$;
   rules :
   $a : A \rightarrow b : A \quad [\textbf{PosFrom}(c) == b.val] \quad c : line \quad [\textbf{PosTo}(c) == d.val] \quad d : square \quad \{a.val.pos = d.val.pos\}$
   $\mid \quad b : square \quad \{a.val.pos = b.val.pos\}$;
   }

   Cet exemple décrit la syntaxe d'un ensemble de polygones de forme carré, reliés par des segments. Une section nomme et définit le typage des terminaux (lexèmes), une autre précise similairement les non-terminaux, et enfin une dernière section donne les règles de la grammaire, avec les prédicats relationnels (entre crochets) et l'affectation des attributs (entre accolades).

2. *Prédicats relationnels.* Il est commode de fournir un certain nombre de fonctions et prédicats relationnels, qui reviennent de manière récurrente dans les grammaires visuelles. Le tableau de la figure 8.4 définit les principales fonctions et prédicats proposés. Il serait intéressant de mettre en relation ces prédicats avec les logiques spatiales, utilisés dans [53, 56] pour raisonner sur la syntaxe et sémantique des langages visuels. De nombreuses autres relations, non spatiales, sont utilisables pour définir les syntaxes visuelles (couleur, forme, dimension).

fonctions	type	calcul	signification
GetVec	$S\_Glyph \rightarrow Pos$	$rot(g.orient, \overrightarrow{V}) * g.size)$	vecteur caractéristique
PosFrom	$G\_Line \rightarrow Pos$	$sub(g.pos, GetVec(g))$	origine du segment
PosTo	$G\_Line \rightarrow Pos$	$add(g.pos, GetVec(g))$	destination du segment
Icircle	$G\_Polygon \rightarrow S\_Glyph$	$\langle pos = g.pos, size = min(g.shape.x_i) * g.shape.norm * g.size \rangle$	cercle inscrit
Dist	$Pos \times Pos \rightarrow num$	$\sqrt{(g_2.pos.x - g_1.pos.x)^2 + (g_2.pos.y - g_1.pos.y)^2}$	dist. euclidienne
prédicats	**type**	**calcul**	**signification**
OnRight	$Pos \times Pos \rightarrow bool$	$g_1.pos.x > g_2.pos.x$	g_1 à droite de g_2
OnLeft	$Pos \times Pos \rightarrow bool$	$g_1.pos.x < g_2.pos.x$	g_1 à droite de g_2
XAlign	$Pos \times Pos \rightarrow bool$	$g_1.pos.x == g_2.pos.x$	g_1 et g_2 alignés en X
YAlign	$Pos \times Pos \rightarrow bool$	$g_1.pos.y == g_2.pos.y$	g_1 et g_2 alignés en Y
Above	$Pos \times Pos \rightarrow bool$	$g_1.pos.y < g_2.pos.y$	g_1 au dessus de g_2
Below	$Pos \times Pos \rightarrow bool$	$g_1.pos.x > g_2.pos.x$	g_1 au dessous de g_2
Inside	$Pos \times Glyph \rightarrow bool$	$Dist(g_1, g_2) < Icircle(g_2).size$	g_1 à l'intérieur de g_2
Outside	$Pos \times Glyph \rightarrow bool$	$Dist(g_1, g_2) > g_2.size$	g_1 à l'extérieur de g_2
Near	$S\_Glyph \times S\_Glyph \rightarrow bool$	$Dist(g_1, g_2) \leq g_2.size + g_1.size$	g_1 proche de g_2
Overlapp	$G\_Polygon \times G\_Polygon \rightarrow bool$	$Dist(g_1, g_2) < Icircle(g_1).size + Icircle(g_2).size$	g_1 recouvre g_2
Contain	$G\_Polygon \times S\_Glyph \rightarrow bool$	$Dist(g_1, g_2) + g_2.size \leq Icircle(g_1).size$	g_1 contient g_2
Beside	$S\_Glyph \times S\_Glyph \rightarrow bool$	$Dist(g_1, g_2) > g_1.size + g_2.size$	g_1 à coté de g_2

FIG. 8.4 : Les prédicats et fonctions de base

Reconnaissance des phrases visuelles Ici encore, un parser spécial interprète un ensemble de lexèmes au moyen des tables "action", "goto" et "next", en suivant le même principe : favoriser la composition ultérieure. Toutefois, la façon dont les lexèmes d'entrée sont recherchés diffère de la technique classique (acquisition en séquence) : c'est le *syntax directed input scanning* évoqué en [38]. La table "next" contient, pour chaque état et chaque type de terminal le prédicat que ce dernier doit vérifier avec les opérandes contenues dans la pile. Les lexèmes qui n'ont pas encore été analysés sont "essayés" jusqu'a ce qu'un terminal soit trouvé. Dans le cas contraire, une erreur de syntaxe est signalée.

8.5 Instructions graphiques

Les figures 8.5,8.6,8.7,8.8 ,8.9 et 8.10 décrivent le jeu d'instructions reconnu par la machine graphique. Les signatures expriment les opérations de piles au moyen d'une règle de réécriture (S symbolise la pile). On distingue plusieurs groupes logiques :

1. *opérations générales.* Ce sont les instructions de manipulation de la pile, des constructeurs de listes et d'ensembles, des opérations arithmétiques polymorphes sur les nombres, matrices et ensembles. De plus, une instruction permet le calcul de la distance euclidienne. Notons que le polymorphisme apporté par le sous-typage de circus simplifie le jeu d'instruction : $DIST$, par exemple, peut-être appliqué à une opérande de type Pos ou de type $Glyph$ ou $G\_Polygon$ et autres sous-types.

2. *constructeurs de matrices et transformations.* Les matrices considérées sont de type 3×3 à coefficients réels. Elle permettent de décrire des rotations, homothéties et translations, ou leurs composées dans un espace 2D. Typiquement,

code	*commentaires* action sur la pile	types
PUSH(v)	*empile une valeur* v	
	$S \rightarrow S, v$	$v : \top$
BUILD_SET	*construit un ensemble* ($n > 0$)	
	$S, v_1, \cdots, v_n, n \rightarrow S, \{v_1, \cdots, v_n\}$	$v_i : \top, n :$ **num**
BUILD_SEQ	*construit une liste* ($n > 0$)	
	$S, v_1, \cdots, v_n, n \rightarrow S, [v_1, \cdots, v_n]$	$v_i : \top, n :$ **num**, $[v_i] : [\top]$
POP	*détruit le sommet de la pile*	
	$S, v \rightarrow S$	$v : \top$
DUP	*duplique le sommet de la pile*	
	$S, v \rightarrow S, v, v$	$v : \top$
MUL_NN	*multiplication numérique*	
	$S, v_1, v_2 \rightarrow S, v_1 \times v_2$	$v_1 :$ **num**, $v_2 :$ **num**, $v_1 \times v_2 :$ **num**
MUL_PN	*produit scalaire/vecteur*	
	$S, p_1, v \rightarrow S, p_1 \times v$	$p_1 : Pos, v :$ **num**, $p_1 \times v : Pos$
MUL_MM	*mult. matricielle*	
	$S, m_1, m_2 \rightarrow S, m_1 \times m_2$	$m_1 : Mat, m_2 : Mat, m_1 \times m_2 : Mat$
DIV_NN	*division numérique*	
	$S, v_1, v_2 \rightarrow S, v_1 / v_2$	$v_1 :$ **num**, $v_2 :$ **num**, $v_1 / v_2 :$ **num**
DIV_PN	*div. scalaire/vecteur*	
	$S, p, v \rightarrow S, p / v$	$p : Pos, v :$ **num**, $p / v : Pos$
ADD_NN	*addition numérique*	
	$S, v_1, v_2 \rightarrow S, v_1 + v_2$	$v_1 :$ **num**, $v_2 :$ **num**, $v_1 + v_2 :$ **num**
ADD_PP	*addition vectorielle*	
	$S, p_1, p_2 \rightarrow S, p_1 + p_2$	$p_1 : Pos, p_2 : Pos, p_1 + p_2 : Pos$
ADD_SET	*addition d'ensembles (inclusion)*	
	$S, s_1, s_2 \rightarrow S, s_1 + s_2$	$s_1 : \{Pos\}, s_2 : \{Pos\}, s_1 + s_2 : \{Pos\}$
SUB_NN	*soustraction numérique*	
	$S, v_1, v_2 \rightarrow S, v_1 - v_2$	$v_1 :$ **num**, $v_2 :$ **num**, $v_1 - v_2 :$ **num**
SUB_PP	*soustraction vectorielle*	
	$S, p_1, p_2 \rightarrow S, p_1 - p_2$	$p_1 : Pos, p_2 : Pos, p_1 - p_2 : Pos$
SUB_SET	*soustraction d'ensembles*	
	$S, s_1, s_2 \rightarrow S, s_1 - s_2$	$s_1 : \{Pos\}, s_2 : \{Pos\}, s_1 - s_2 : \{Pos\}$
NEG_N	*inversion numérique*	
	$S, v \rightarrow S, -v$	$v :$ **num**, $-v :$ **num**
NEG_P	*mult. scalaire par -1*	
	$S, p \rightarrow S, -p$	$p : Pos, -p : Pos$
DIST	*distance euclidienne*	
	$S, p_1, p_2 \rightarrow S, d$	$p_1 : Pos, p_2 : Pos, d :$ **num**

FIG. 8.5 : Le code opérationnel de la machine graphique (partie 1)

code	*commentaires*	
	action sur la pile	types
MATRIX_POS	*constr. matrice de translation*	
	$S, p \rightarrow S, m$	$p : Pos, m : Mat$
MATRIX_ROT	*constr. matrice de rotation*	
	$S, p, \theta \rightarrow S, m$	$p : Pos, \theta :$ **num**, $m : Mat$
MATRIX_HOM	*constr. matrice d'homothétie*	
	$S, p, k \rightarrow S, m$	$p : Pos, k :$ **num**, $m : Mat$
TRANS	*séquence de transformations*	
	$S, \{g_i\}, [m_k] \rightarrow S$	$\{g_i\} : \{S\_Glyph\}, [m_k] : [Mat]$
GET_TIME	*temps logique (réel, en secondes)*	
	$S \rightarrow S, t$	$t :$ **num**
ANIMATE	*animation lancée en t durant dt*	
	$S, g, m, t, dt \rightarrow S$	$g : S\_Glyph, m : Mat, t :$ **num**, $dt :$ **num**
CREATE_LEX	*crée un type lexical s*	
	$S, t, s \rightarrow S$	$t : type, s :$ **string**, $t \preceq S\_Glyph$
CREATE	*instancie un type lexical*	
	$S, s \rightarrow S, g$	$s :$ **string**, $g : \tau, \tau \preceq Pos$
DELETE	*détruit un glyphe*	
	$S, g \rightarrow S$	$g : \tau, \tau \preceq Pos$
COPY	*crée un glyphe identique*	
	$S, g \rightarrow S, g, g$	$g : t, \tau \preceq Pos$
HIDE	*rend un glyphe invisible*	
	$S, g \rightarrow S$	$g : Glyph$
SHOW	*rend un glyphe visible*	
	$S, g \rightarrow S$	$g : Glyph$
SENS	*rend un glyphe sensible au pointeur*	
	$S, g \rightarrow S$	$g : Glyph$
UNSENS	*rend un glyphe insensible au pointeur*	
	$S, g \rightarrow S$	$g : Glyph$
UPPER	*g_1 sera dessiné au-dessus de g_2*	
	$S, g_1, g_2 \rightarrow S$	$g_1 : Glyph, g_2 : Glyph$
LOWER	*g_1 sera dessiné au-dessous de g_2*	
	$S, g_1, g_2 \rightarrow S$	$g_1 : Glyph, g_2 : Glyph$
ONTOP	*g sera dessiné au-dessus de tous*	
	$S, g \rightarrow S$	$g : Glyph$
ONBOTTOM	*g sera dessiné au-dessous de tous*	
	$S, g \rightarrow S$	$g : Glyph$

FIG. 8.6 : Le code opérationnel de la machine graphique (partie 2)

code	*commentaires*	
	action sur la pile	types
POS/POS_TO	*déplacement relatif/absolu d'un vecteur* v	
	$S, p, v \rightarrow S$	$p : Pos, v : A\_pos$
SIZE/SIZE_TO	*redimensionnement relatif/absolu*	
	$S, g, s \rightarrow S$	$g : S\_Glyph, s : A\_size$
SHAPE/SHAPE_TO	*morphing relatif/absolu*	
	$S, g, s \rightarrow S$	$g : G\_Polygon, s : A\_shape$
COLOR/COLOR_TO	*coloration relative/absolue*	
	$S, g, c \rightarrow S$	$g : C\_Glyph, c : A\_color$
ORIENT/ORIENT_TO	*réorientation relative/absolue*	
	$S, g, o \rightarrow S$	$g : Glyph, o : A\_orient$
TEXT	*insert* s *dans le glyphe* g	
	$S, g, s \rightarrow S$	$g : Text, s :$ **string**
TEXT_TO	*insert* s *à la position* i	
	$S, g, s, i \rightarrow S$	$g : Text, s :$ **string**, $i :$ **num**

FIG. 8.7 : Le code opérationnel de la machine graphique (partie 3)

code	*commentaires*	
	action sur la pile	types
GET_POS	*position du glyphe*	
	$S, g \rightarrow S, p$	$g : Pos, p : A\_pos$
GET_SIZE	*dimension du glyphe*	
	$S, g \rightarrow S, s$	$g : S\_Glyph, s : A\_size$
GET_SHAPE	*forme du glyphe*	
	$S, g \rightarrow S, s$	$g : G\_Polygon, s : A\_shape$
GET_COLOR	*couleur du glyphe*	
	$S, g \rightarrow S, c$	$g : C\_Glyph, c : A\_color$
GET_ORIENT	*orientation du glyphe*	
	$S, g \rightarrow S, o$	$g : Glyph, o : A\_orient$
GET_ICIRC	*cercle inclus*	
	$S, g_1 \rightarrow S, g_2$	$g_1 : S\_Glyph, g_2 : G\_Circle$
GET_BCIRC	*cercle englobant*	
	$S, g_1 \rightarrow S, g_2$	$g : S\_Glyph, g_2 : G\_Circle$
GET_TEXT	*texte contenu dans le glyphe*	
	$S, g \rightarrow S, s$	$g : Text, s :$ **string**

FIG. 8.8 : Le code opérationnel de la machine graphique (partie 4)

code	*commentaires*	
	action sur la pile	types
GET_MOUSE	*pointeur de la souris*	
	$S \rightarrow S, g$	$g : S\_Glyph$
SET_MOUSE	*change le pointeur de souris*	
	$S, g \rightarrow S$	$g : S\_Glyph$
GET_CONTENT	*glyphes spatialement inclus*	
	$S, g \rightarrow S, \{g_i\}$	$g : S\_Glyph, \{g_i\} : \{S\_Glyph\}$
GROUP	*crée un glyphe composite*	
	$S, \{g_i\} \rightarrow S, g$	$\{g_i\} : \{Glyph\}, g : CompoundGlyph$
UN_GROUP	*décompose un glyphe composite*	
	$S, g \rightarrow \{g_i\}$	$\{g_i\} : \{Glyph\}, g : CompoundGlyph$
STORE	*mémorise une valeur*	
	$S, g, s, v \rightarrow S$	$g : Glyph, s :$ **string**, $v : \top$
READ	*lit une valeur*	
	$S, g, s \rightarrow S, v$	$g : Glyph, s :$ **string**, $v : \top$
GET	*lit et détruit une valeur*	
	$S, g, s \rightarrow v$	$g : Glyph, s :$ **string**, $v : \top$
FILTER_POS	*filtre un ensemble de glyphe par leur position (e écart toléré)*	
	$S, p, d, \{g_i\} \rightarrow S, \{g_k\}$	$g : A\_pos, e :$ **num**, $\{g_k\} \subset \{g_i\}$
FILTER_SIZE	*filtre un ensemble de glyphe par leur dimension (e écart toléré)*	
	$S, d, e, \{g_i\} \rightarrow S, \{g_k\}$	$d :$ **num**, $e :$ **num**, $\{g_k\} \subset \{g_i\}$
FILTER_ORIENT	*filtre un ensemble de glyphe par leur orientation (e écart toléré)*	
	$S, o, e, \{g_i\} \rightarrow S, \{g_k\}$	$o :$ **num**, $e :$ **num**, $\{g_k\} \subset \{g_i\}$
FILTER_LEX	*filtre un ensemble de glyphe par leur type lexical*	
	$S, l, \{g_i\} \rightarrow S, \{g_k\}$	$l :$ **string**, $\{g_k\} \subset \{g_i\}$
GET_TYPE	*retourne le type d'un glyphe*	
	$S, g \rightarrow S, t$	$g : \tau, \tau \preceq Pos, t :$ **type**
EQ_TYPE	*teste l'égalité de types*	
	$S, t_1, t_2 \rightarrow S, r$	$r :$ **bool**, $t_1 :$ **type**, $Term t_2$ **type**
SUB_TYPE	*teste si t_1 est un sous-type de t_2*	
	$S, t_1, t_2 \rightarrow S, r$	$r :$ **bool**, $t_1 :$ **type**, $Term t_2$ **type**

FIG. 8.9 : Le code opérationnel de la machine graphique (partie 5)

code	*commentaires*	
	action sur la pile	types
SET_CSTR_POS	*contrainte de positionnement (Pos : decalage)*	
	$S, g, \{p_i\}, p \rightarrow S$	$g : Pos, \{p_i\} : \{Pos\}, p : Pos$
SET_CSTR_SIZE	*contrainte de dimensionnement*	
	$S, g, \{S\_Glyph\}, s \rightarrow S$	$g : S\_Glyph, \{g_i\} : \{S\_Glyph\}, s : A\_size$
SET_CSTR_ORIENT	*contrainte d'orientation*	
	$S, g, \{g_i\}, o \rightarrow S$	$g : Glyph, \{g_i\} : \{Glyph\}, o : A\_orient$
GET_CSTR_POS	*ensemble de glyphes contraints par g_1 (position)*	
	$S, g \rightarrow S, \{g_i\}$	$g : Pos, \{g_i\} : \{Pos\}$
GET_CSTR_SIZE	*ensemble de glyphes contraints par g_1 (dimension)*	
	$S, g \rightarrow S, \{g_i\}$	$g : S\_Glyph, \{g_i\} : \{S\_Glyph\}$
GET_CSTR_ORIENT	*ensemble de glyphes contraints par g_1 (orientation)*	
	$S, g \rightarrow S, \{g_i\}$	$g : Glyph, \{g_i\} : \{Glyph\}$
GET_CSTR	*ensemble de glyphes contraints par g*	
	$S, g \rightarrow S, \{g_i\}$	$g : S\_Glyph, \{g_i\} : \{S\_Glyph\}$
UNSET_CSTR	*suppr. toutes contraintes appliquées sur les g_i*	
	$S, \{g_i\} \rightarrow S$	$\{g_i\} : \{S\_Glyph\}$
UNSET_CSTR_POS	*suppr. toutes contraintes de positionnement*	
	$S, \{g_i\} \rightarrow S$	$\{g_i\} : \{Pos\}$
UNSET_CSTR_SIZE	*suppr. toutes contraintes de dimensionnement*	
	$S, \{g_i\} \rightarrow S$	$\{g_i\} : \{S\_Glyph\}$
UNSET_CSTR_ORIENT	*suppr.toutes contraintes d'orientation*	
	$S, \{g_i\} \rightarrow S$	$\{g_i\} : \{Glyph\}$

FIG. 8.10 : Le code opérationnel de la machine graphique (partie 6)

Rotation d'angle θ $\quad \mathcal{R}_\theta = \begin{pmatrix} \mathbf{cos}\theta & -\mathbf{sin}\theta & 0 \\ \mathbf{sin}\theta & \mathbf{cos}\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Homothétie de rapport k $\quad \mathcal{H}_k = \begin{pmatrix} k & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Translation d'un vecteur $v : Pos$ $\quad \mathcal{T}_v = \begin{pmatrix} 1 & 0 & v.x \\ 0 & 1 & v.y \\ 0 & 0 & 1 \end{pmatrix}$

coordonnées d'une variable $p : Pos$ $\quad \mathcal{P}_p = \begin{pmatrix} p.x \\ p.y \\ 1 \end{pmatrix}$

La composition de transformations est exprimée par multiplication des matrices : une translation T de matrice $\mathcal{T}_v$ et une homothétie H, de matrice $\mathcal{H}_k$,

$$T \circ H = \mathcal{T}_v \times \mathcal{H}_k = \begin{pmatrix} k & 0 & v.x \\ 0 & k & v.y \\ 0 & 0 & 1 \end{pmatrix}$$

L'application d'une transformation M de matrice $\mathcal{M}$ à un point $p : Pos$ est :

$$M(p) = \mathcal{P}_p \times \mathcal{M} = \begin{pmatrix} p.x \\ p.y \\ 1 \end{pmatrix} \times \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} p.x \times m_{11} + p.y \times m_{12} + m_{13} \\ p.x \times m_{21} + p.y \times m_{22} + m_{23} \\ 1 \end{pmatrix}$$

L'interêt de ce système de représentation des coordonnées, appellé représentation en coordonnées homogènes, est de ramener les transformations de l'espace à une opération simple (multiplication

de matrices) et à un format unique pour coder la transformation.

3. *primitives d'animations.* Une animation commence à un temps absolu, et dure un intervalle de temps fixe. Elle réalise une transformation spatiale ou autre en applicant une accélération calculée en interne. Cette accélération permet de préserver la continuité perceptuelle, tout en optimisant la durée [1] (cf [100, 118, 109]). La machine visuelle gère la cohérence entre la liste d'animation et la destruction des glyphes.
4. *constructeurs-destructeurs de glyphes.* Un glyphe possède toujours un type "lexical", utilisé pour la génération des lexèmes d'action et l'analyse de la syntaxe visuelle. Les glyphes peuvent être cachés à l'utilisateur, bien que présent dans la liste interne d'affichage (Ils sont visibles par défaut). De même, ils peuvent être rendus insensibles au pointeur de souris (aucun lexème d'interaction émis lorsque le pointeur entre ou quitte l'objet graphique). Il est également possible de changer l'ordre d'affichage (permet de dessiner un polygone au-dessus d'un autre polygone par exemple)
5. *modification des attributs perceptuels.* D'une manière générale, la modification est relative ou absolue.
6. *consultation des attributs perceptuels.* Opérations symétriques des précédentes.
7. *opérations sur l'environnement.* elles permettent de : positionner et connaître les paramètres liés au pointeur de souris (considéré comme un glyphe) ; connaître l'ensemble des glyphes (spatialement) contenus dans un autre glyphe ; grouper un ensemble de glyphes dans un unique glyphe composite (facilite la manipulation), et de faire l'inverse ; associer des informations à un glyphe quelconque.
8. *contraintes.* Il est possible de spécifier et retracter des contraintes. Ces dernières sont orientées , d'un glyphe vers un groupe de glyphes. Ces contraintes peuvent porter sur tous les attributs spatiaux. Elle consistent à maintenir constante une valeur de décalage. Les cycles sont gérés par le solveur. Les cycles imcompatibles sont détectés et signalés (la somme des valeurs de décalage dans le cycle doit être nulle). Lorsqu'un glyphe est détruit, toutes les contraintes qui lui sont rattachées sont détruites également. C'est une approche très simple des contraintes spatiales, suffisante pour gérer la cohérence des glyphes composites, mais qui pourrait être étendue en bénéficiant des nombreux travaux de recherche menés dans ce domaine.

[1] Il devient en fait possible de spécifier des animation de courte durée, qui reste très perceptibles grâce à l'accélération progressive.

Le langage *While* décrit en *Circus* : version textuelle

Nous présentons une version enrichie de *While*, acceptant des déclarations de types simplifiées et des opérations d'indexage. Les types primitifs traités sont les entiers et les chaînes de caractères ; les types structurés sont des tableaux. Bien que simple, cette approche permet d'illustrer de façon réaliste les problèmes génériques liés à la réalisation de compilateurs, et les solutions proposées en *Circus*. Certains des composants sont mis en commun dans la version textuelle et visuelle du langage.

L'architecture proposée est "classique" : une partie frontale, constituée d'un analyseur lexical, génère une séquence de symboles, un analyseur syntaxique construit un arbre concret (décoré avec les symboles lexicaux), puis un transducteur construit un arbre de syntaxe abstrait. La partie centrale est constituée d'un contrôleur de types, et la partie "arrière" se compose d'un générateur de code pouvant être interprété par une machine virtuelle, elle-même décrite en *Circus*.

9.1 Analyse syntaxique

La grammaire de *While* est décrite au moyen de plusieurs modules, correspondant à la décomposition naturelle des éléments syntaxiques :

1. Structure générale.

```
grammar WhileM {
⟨L⟩ → ⟨C⟩ ';' ⟨L⟩ | ⟨C⟩;
⟨C⟩ → id ':=' ⟨A⟩
    | 'if' ⟨B⟩ 'then' ⟨C⟩ 'else' ⟨C⟩
    | 'while' ⟨B⟩ 'do' ⟨C⟩ 'end'
    | 'skip' ;
}
```

2. Expressions booléennes.

```
grammar WhileE {
⟨B⟩ → 'true' | 'false' | 'not' ⟨B⟩
    | '(' ⟨B⟩ 'and' ⟨B⟩ ')'
    | '(' ⟨A⟩<⟨A⟩ ')'
    | '(' ⟨A⟩ '=' ⟨A⟩ ')' ;
}
```

3. Expressions arithmétiques.

```
grammar WhileA {
⟨A⟩ → n                          // unité lexicale numérique
    | s                          // unité lexicale chaîne de caractères
    | id                         // identificateur
    | '(' ⟨A⟩ ⟨op⟩ ⟨A⟩ ')' ;
⟨op⟩ → '*' | '+' | '-' | '/' ;
}
```

4. Types.

```
grammar WhileT {
⟨T⟩ → 'num' | 'string' | 'boolean'
    | 'array' '[' n ']' 'of' ⟨T⟩;
}
```

La grammaire finale est obtenue en utilisant les facilités de composition offertes par *Circus*, et en ajoutant les règles assurant la liaison avec la syntaxe des types

$$
\begin{array}{lcl}
\textbf{grammar}\, While & : & WhileA, WhileE, WhileM, WhileT\{ \\
\langle C \rangle & +\!\!\rightarrow & \textbf{id}\ \text{' :'}\ \textbf{T} \\
 & ; & \\
\langle A \rangle & +\!\!\rightarrow & \textbf{id}\ \langle S \rangle \\
 & ; & \\
\langle S \rangle & \rightarrow & \text{'[' } \langle A \rangle \text{ ']'} \\
 & | & \text{'[' } \langle A \rangle \text{ ']' } \langle S \rangle \\
 & ; & \\
\} & &
\end{array}
$$

9.2 Convertisseur en syntaxe abstraite

La syntaxe abstraite est plus simple que la syntaxe concrète. Elle isole les traitements sémantiques des modifications légères susceptibles d'être apportées à la grammaire initiale. De plus, en simplifiant l'arbre, elle simplifie et rend plus efficace son traitement. L'arbre syntaxique abstrait est construit par une machine qui comporte une collection de règles, avec en partie gauche un filtrage de l'arbre concret et en partie droite le code qui génere l'arbre abstrait. Chaque règle correspond à un noeud de la grammaire concrète.

La machine abstraite qui réalise la transformation (nommée $MAST$) possède le squelette suivant

$$
\begin{array}{lcl}
\textbf{type}\ Anode & = & \langle node : \top, sub : [\top] \rangle \\
 & & \\
\textbf{const} MAST_3 & := & \nabla x : [\top],\ \textbf{y}\ Anode. \\
 & = & \{ \\
 & & R\_while : \cdots ; \\
 & & R\_aff : \cdots ; \\
 & & R\_skip : \cdots ; \\
 & & R\_tst : \cdots ; \\
 & & R\_decl : \cdots ; \\
 & & \vdots \\
 & & \}
\end{array}
$$

Notons qu'il est possible de décomposer le traitement de manière modulaire (par exemple, parallèlement à la décomposition modulaire de la grammaire) en ecrivant plusieurs machines et en les composant avec **Then** .

9.3 Contrôleur de types

Le contrôle de type, dans le mini-langage etudié, implémente le système suivant :

1. Typage des unité lexicales.

$$\frac{}{\gamma \rhd \textbf{true} : \textbf{bool}} \quad \frac{}{\gamma \rhd \textbf{false} : \textbf{bool}} \quad \frac{}{\gamma \rhd \textbf{n} : \textbf{num}} \quad \frac{}{\gamma \rhd \textbf{s} : \textbf{string}}$$

2. Typage des opérateurs.

$$\frac{\gamma \rhd v_1 : num \quad \gamma \rhd v_2 : num}{\gamma \rhd v_1 \star v_2 : num} \star \in \{+, *, -, /\} \qquad \frac{\gamma \rhd v_1 : string \quad \gamma \rhd v_2 : string}{\gamma \rhd v_1 + v_2 : string}$$

$$\frac{\gamma \rhd v_1 : num \quad \gamma \rhd v_2 : num}{\gamma \rhd v_1 \star v_2 : bool} \star \in \{<, =\} \qquad \frac{\gamma \rhd v_1 : bool \quad \gamma \rhd v_2 : bool}{\gamma \rhd v_1 \ \textbf{and}\ v_2 : bool}$$

$$\frac{\gamma \rhd v_1 : bool}{\gamma \rhd \ \textbf{not}\ v_1 : bool} \qquad \frac{\gamma \rhd v_1 : \ \textbf{array}\,[n]\ \textbf{of}\ t_1 \quad \gamma \rhd v_2 : num}{\gamma \rhd v_1[v_2] : t_1}$$

3. Typage des constructeurs.

$$\frac{\gamma \rhd B : bool \quad \gamma \rhd C : unit \quad \gamma \rhd C : unit\}}{\gamma \rhd \ \textbf{while}\ B\ \textbf{do}\ C_1\ \textbf{;}\ C_2 : unit} \qquad \frac{\gamma \rhd B : bool \quad \gamma \rhd C_1 : unit \quad \gamma \rhd C_2 : unit}{\gamma \rhd \ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 : unit}$$

$$\frac{\textbf{id} \notin dom(\gamma) \quad \gamma \vdash type(T)}{\gamma \rhd \ \textbf{id}\ \textbf{:}\ T : unit} \qquad \frac{\gamma \rhd id : t \quad \gamma \rhd E : t}{\gamma \rhd \ \textbf{id}\ \textbf{:=}\ E : unit}$$

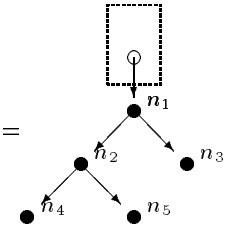
$$\frac{\gamma \rhd C_1 : unit \quad \gamma \rhd C_2 : unit}{\gamma \rhd C_1\ \textbf{;}\ C_2 : unit} \qquad \frac{\gamma \rhd \ \textbf{id}\ \textbf{:}\ T : unit \quad \gamma, id : type(T) \vdash C_2 : unit}{\gamma \rhd (\ \textbf{id}\ \textbf{:}\ T)\ \textbf{;}\ C_2 : unit}$$

unit est un type "virtuel" permettant de traiter de manière homogène les constructions impératives (cf [23]).

Ce système de type est traité par des machines travaillant sur l'arbre syntaxique abstrait. La première d'entre elles, $CompType$, réalise la décoration des sous-arbres représentant les expressions arithmétiques. Elle utilise la mémoire globale pour recevoir les expressions à évaluer, et transmettre le résultat. Pour parcourir l'arbre abstrait de manière générique, une machine spécialisée, P_EXP, calcule la liste des nœuds à parcourir (profondeur d'abord, en remontant des feuilles vers la racine ; les feuilles sont détectés car leur sous arbre (champ "sub") est vide)

$$\begin{array}{lll}
\textbf{P_EXP} & := & \nabla x : [Anode], y : \langle nodes : [\top] \rangle. \\
 & & \textbf{var}\ n : \top.. \\
 & & \textbf{var}\ m : \top.. \\
 & & \textbf{var}\ s : [Anode].. \\
 & & \textbf{var}\ L : [Anode].. \\
 & & \{ \\
 & & \quad r1 : x \ \sharp\ [\langle node = ?n, sub = [?m] \boxplus ?s \rangle] \boxplus ?L \\
 & & \qquad \Rightarrow x = [m] + s + L; y.nodes = y.nodes + [n] \\
 & & \quad r2 : x \ \sharp\ [\langle node = ?n, sub = \%[\,] \rangle] \boxplus ?L \\
 & & \qquad \Rightarrow x = L; y.nodes = y.nodes + [n] \\
 & & \}
\end{array}$$

L'exemple suivant illustre le travail de la machine sur le sous-arbre abstrait correspondant à l'expression $(a + 1) * b$, c'est à dire :

$G =$ 

$$
\begin{array}{lcl}
n_1 & = & \langle val = 0, op =' *' \rangle \\
n_2 & = & \langle val = 0, op =' +' \rangle \\
n_3 & = & \langle val =' b', op =' id' \rangle \\
n_4 & = & \langle val =' a', op =' id' \rangle \\
n_5 & = & \langle val = 1, op =' cst' \rangle
\end{array}
$$

L'expression textuelle est

$$G = \langle node = n_1, sub = [\langle node = n_2, sub = [\langle node = n_4, sub = [] \rangle, \langle node = n_5, sub = [] \rangle] \rangle, \langle node = n_3, sub = [] \rangle]$$

Nous produisons la dérivation complète, avec les états intermédiaires de la machine lorsque ces derniers sont modifiés (chaque dérivation correspond à l'application d'une règle) :

$$
\begin{array}{ll}
 & \langle x = [G] \rangle, \langle node = \textbf{none} \rangle \\
\stackrel{r_1}{\Longrightarrow} & \langle x = [\langle node = n_2, sub = [\langle node = n_4, sub = [] \rangle, \langle node = n_5, sub = [] \rangle] \rangle, \langle node = n_3, sub = [] \rangle] \rangle, \langle nodes = [n_1] \rangle \\
\stackrel{r_1}{\Longrightarrow} & \langle x = [\langle node = n_4, sub = [] \rangle, \langle node = n_5, sub = [] \rangle, \langle node = n_3, sub = [] \rangle] \rangle, \langle nodes = [n_2, n_1] \rangle \\
\stackrel{r_2}{\Longrightarrow} & \langle x = [\langle node = n_5, sub = [] \rangle, \langle node = n_3, sub = [] \rangle] \rangle, \langle nodes = [n_4, n_2, n_1] \rangle \\
\stackrel{r_2}{\Longrightarrow} & \langle x = [\langle node = n_3, sub = [] \rangle] \rangle, \langle nodes = [n_5, n_4, n_2, n_1] \rangle \\
\stackrel{r_2}{\Longrightarrow} & \langle x = [] \rangle, \langle nodes = [n_3, n_5, n_4, n_2, n_1] \rangle
\end{array}
$$

La vérification des types est réalisée par une série de machines, chacune ayant en charge le contrôle d'une des constructions syntaxiques de *While*. Elle sont toutes composées avec P_EXP de la manière suivante.

$$\textbf{const}\ ProcTypes := P\_EXP\ \textbf{Then}\ (CheckId\ \textbf{Then}\ CheckCst \cdots CheckAff\ \textbf{Then}\ CheckDecl)$$

Lorsque l'ensemble de l'arbre abstrait est exploré par $ProcTypes$, toutes les erreurs de type sont détectées, et l'arbre abstrait est décoré avec un attribut qui contient l'encodage du type (sous forme d'une chaîne). Les messages d'erreurs potentiellement générés sont :

```
'bad operator for numeric operands'
'bad operator for string operands (``+'' expected)'
'bad operator for array operands (``[ ]'' expected)'
'undefined reference'
'incompatible operands'
'reference already defined'
```

9.4 Interpréteur

Celui-ci utilise une pile $St : [\top]$, une mémoire globale $Mem : \{num : \top]\}$, et utilise un segment de constante $Cseg : \{num : \top\}$. L'interpréteur est composé d'une unité de calcul ALU, d'une unité de gestion mémoire MU et d'une unité gérant les débranchements BU. Elle interprète un code $Code : [\top]$ constitué d'une liste numérique d'instructions et d'opérandes au moyen d'un pointeur d'instruction $Ip : num$.

const ADD : **num** $= 20$ **const** SUB : **num** $= 21$ **const** MUL : **num** $= 22$ **const** DIV : **num** $= 23$
type $ALU\_OP =$ **num in** $\{ADD, SUB, MUL, DIV, CMP, NOP\}$;
type $ALU\_sig_2 = \langle Mem : \{num : \top]\}, St : [\top]\rangle$;
type $ALU\_sig_1 = \langle Ip : num, Code : [ALU\_OP], Cseg : \{num : \top]\}\rangle$;

type $T\_ALU = ALU\_sig_1 \Longrightarrow ALU\_sig_2$

const ALU : $T\_ALU = \nabla x : ALU\_sig_1, y : ALU\_sig_2.$
var $st : [\top].$ **var** v_1 : **num** . **var** $v_2 : num.$
{
$nop : x.Code[x.Ip : x.Ip + 1] \not\Vdash [\%NOP] \Rightarrow Ip = Ip + 1,$
$add :$
$x.Code[x.Ip : x.Ip + 1] \not\Vdash [\%ADD] \Rightarrow$
$y \not\Vdash \langle St = ?st + [?v_1, ?v_2]\rangle \Rightarrow y.St = st + [v_1 + v_2]; x.Ip = x.Ip + 1,$
$sub :$
$x.Code[x.Ip : x.Ip + 1] \not\Vdash [\%SUB] \Rightarrow$
$y \not\Vdash \langle St = ?st + [?v_1, ?v_2]\rangle \Rightarrow y.St = st + [v_1 - v_2]; x.Ip = x.Ip + 1,$
$mul :$
$x.Code[x.Ip : x.Ip + 1] \not\Vdash [\%MUL] \Rightarrow$
$y \not\Vdash \langle St = ?st + [?v_1, ?v_2]\rangle \Rightarrow y.St = st + [v_1 * v_2]; x.Ip = x.Ip + 1,$
$div :$
$x.Code[x.Ip : x.Ip + 1] \not\Vdash [\%DIV] \Rightarrow$
$y \not\Vdash \langle St = ?st + [?v_1, ?v_2]\rangle \Rightarrow$
if $0 == v_2$ **then** $y.St = st + [0]$ **else** $y.St = st + [v_1/v_2];$
$x.Ip = x.Ip + 1,$
$cmp :$
$x.Code[x.Ip : x.Ip + 1] \not\Vdash [\%CMP] \Rightarrow$
$y \not\Vdash \langle St = ?st + [?v_1, ?v_2]\rangle \Rightarrow$
if $v_1 \leq v_2$ **then** $y.St = st + [0]$ **else** $y.St = st + [1];$
$x.Ip = x.Ip + 1,$

}

const POP : **num** $= 10$; **const** $PUSH$: **num** $= 11$; **const** ROT : **num** $= 12$; **const** DUP : **num** $= 13$;
const $STORE$: **num** $= 14$; **const** GET : **num** $= 15$; **const** $CGET$: **num** $= 16$;
type $MU\_OP =$ **num in** $\{POP, PUSH, ROT, DUP, STORE, GET, CGET\}$;
type $MU\_sig_2 = ALU\_sig_2$;
type $MU\_sig_1 = ALU\_sig_1 \otimes \langle Code : [MU\_OP]\rangle$;

$$
\begin{array}{l}
\textbf{map}\ MU \quad := \quad \nabla x : MU\_sig_1, y : MU\_sig_2 . \\
\quad \textbf{var}\ st : [\top] . \ \textbf{var}\ mem : \{\ \textbf{num}\ : \top\} . \ \textbf{var}\ v_1 : \ \textbf{num}\ . \ \textbf{var}\ v_2 : num . \\
\quad \{ \\
\quad pop : \\
\quad\quad x.Code[x.Ip : x.Ip + 1] \ \natural\ [\%POP] \Rightarrow \\
\quad\quad\quad y \ \natural\ \langle St = ?st + [?v_1] \rangle \Rightarrow y.St = st; x.Ip = x.Ip + 1, \\
\quad push : \\
\quad\quad x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%PUSH, ?v_1] \Rightarrow y.St = y.St + [v_1]; x.Ip = x.Ip + 1, \\
\quad dup : \\
\quad\quad x.Code[x.Ip : x.Ip + 1] \ \natural\ [\%DUP] \Rightarrow \\
\quad\quad\quad y \ \natural\ \langle St = ?st + [?v_1] \rangle \Rightarrow y.St = st + [v_1, v_2]; x.Ip = x.Ip + 1, \\
\quad rot : \\
\quad\quad x.Code[x.Ip : x.Ip + 1] \ \natural\ [\%ROT] \Rightarrow \\
\quad\quad\quad y \ \natural\ \langle St = ?st + [?v_1, ?v_2] \rangle \Rightarrow y.St = st + [v_2, v_1]; x.Ip = x.Ip + 1, \\
\quad store : \\
\quad\quad x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%STORE, ?v_1] \Rightarrow \\
\quad\quad\quad y \ \natural\ \langle St = ?st + [?v_2] \rangle \Rightarrow y.St = st; y.Mem = y.Mem + \{v_1 = v_2\}; x.Ip = x.Ip + 2, \\
\quad get : \\
\quad\quad x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%GET, ?v_1] \Rightarrow \\
\quad\quad\quad y \ \natural\ \langle Mem = ?mem + \{v_1 = ?v_2\} \rangle \Rightarrow y.St = y.St + [v_2]; x.Ip = x.Ip + 2, \\
\quad cget : \\
\quad\quad x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%CGET, ?v_1] \Rightarrow \\
\quad\quad\quad y \ \natural\ \langle Cseg = ?mem + \{v_1 = ?v_2\} \rangle \Rightarrow y.St = y.St + [v_2]; x.Ip = x.Ip + 2, \\
\}
\end{array}
$$

Enfin, l'unité de branchement s'occupe des instructions de "contrôle" : saut absolu JMP, saut relatif JR et relatif conditionnel JE.

$$
\begin{array}{l}
\textbf{const}\ JMP : \ \textbf{num}\ = 30\ \textbf{const}\ JR : \ \textbf{num}\ = 31\ \textbf{const}\ JE : \ \textbf{num}\ = 32 \\
\textbf{type}\ BU\_OP = \ \textbf{num}\ \textbf{in}\ \{JMP, JR, JE\} \\
\textbf{type}\ BU\_sig_2 = ALU\_sig_2 \\
\textbf{type}\ BU\_sig_1 = ALU\_sig_1 \otimes \langle Code : [BU\_OP] \rangle \\
\\
\textbf{const}\ BU \quad := \quad \nabla x : BU\_sig_1, y : BU\_sig_2 . \\
\quad \textbf{var}\ st : [\top] . \ \textbf{var}\ v : \textbf{num} . \ \textbf{var}\ v_2 : \textbf{num} . \\
\quad \{ \\
\quad jmp : x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%JMP, ?v] \Rightarrow x.Ip = v, \\
\\
\quad jr : x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%JR, ?v] \Rightarrow x.Ip = x.Ip + v, \\
\\
\quad je : x.Code[x.Ip : x.Ip + 2] \ \natural\ [\%JMP, ?v] \Rightarrow \\
\quad\quad y \ \natural\ \langle St = ?st + [?v_2] \rangle \Rightarrow \textbf{if}\ v_2 == 0\ \textbf{then}\ x.Ip = x.Ip + 2\ \textbf{else}\ x.Ip = x.Ip + v \\
\}
\end{array}
$$

Finalement, l'interpréteur complet est obtenu par $WVP = \circledast(ALU\ \textbf{Then}\ MU\ \textbf{Then}\ BU)$. Notons qu'un metteur au point peut contrôler la vitesse d'exécution et suivre le déroulement d'un programme *While* par $WVP_2 = \circledast(Sync1\ \textbf{Then}\ ALU(\ \textbf{Then}\ MU\ \textbf{Then}\ BU)\ \textbf{Before}\ Sync2)$, où $Sync$ est telle que

$$
\begin{array}{l}
\textbf{const}\ Sync1 := \nabla x : \langle Ip : \textbf{num} \rangle, y : \langle St : [\top] \rangle . \{ \\
\textbf{var}\ ctx : \langle ip : num, st : [\top] \rangle . \\
\quad tst : \ \textbf{get}\ \%'end' : \textbf{string} \Rightarrow \textbf{none}, \\
\quad sync_1 : \ \textbf{bget}\ ?ctx : \langle ip : \textbf{num}, stack : [\top] \rangle \Rightarrow x.ip = ctx.ip; y.St = ctx.st \\
\} \\
\\
\textbf{const}\ Sync2 := \nabla x : \langle Ip : \textbf{num} \rangle, y : \langle St : [\top] \rangle . \{ \\
\quad sync_2 : \ \textbf{put}\ \langle ip = x.Ip, stack = y.St \rangle : \langle ip : \textbf{num}, stack : [\top] \rangle \\
\}
\end{array}
$$

9.5 Générateur de code

La génération du code doit construire un segment de constantes et une séquence de code. Dans ce dernier, les accès au segment de constante et à la mémoire se font au travers d'adresses précalculées. Ces adresses sont des valeurs numériques. Le générateur que nous présentons travaille en deux passes : la première calcule les adresses et construit des séquences de codes, stokées dans une table indexée par les noeuds de l'arbre abstrait (cf figure 9.1). La deuxième passe concatène les fragments et calcule les déplacements absolus ou relatifs du pointeur d'instruction (cf figure 9.2). Cette passe utilise une machine identique à P_EXP, qui permet un parcours "bottom-up" dans l'ordre postfixe, usuel pour la génération de code destinée aux machines à pile. Il est évident que nous présentons une solution très simplifiée : il serait plus intéressant (mais trop long) de présenter la construction d'un graphe de flot de données et sa transformation. Ou encore, une technique de réécriture d'arbre, basée sur la grammaire des instructions machine, comme décrite en [92] ("BURS", pour *bottom-up rewrite system*), particulièrement efficace.

L'architecture globale du compilateur est clairement exprimée par une expression "algebrique" [1]

$$\textbf{const}\ While := \circledast(Lexer)\ \textbf{Then}\ \circledast(Parser)\ \textbf{Then}\ \circledast(MAST)\ \textbf{Before}$$
$$(TypeControl\ \textbf{During}\ \circledast(ProcType))\ \textbf{Then}\ \circledast(GenCode1\ \textbf{Before}\ GenCode2)$$

C'est certainement une simplification de l'architecture réelle, où la gestion des erreurs vient normalement s'insérer aux différents niveaux du traitement. Toutefois, elle illustre le potentiel de l'approche compositionnelle que nous proposons dans cette étude.

[1] certains composants n'ont pas été détudiés. L'opérateur **During** est extrapolé comme le dual de **Before** , utilisant || au lieu de ;

```
type Taddr = { string : num }
type Tcseg = { num : ⊤}
type TcodeMap = { num : [ num ]}

const opMap : { string : ALU_OP} = {'*' : MUL,'/' : DIV,'+' : ADD,'-' : SUB}

map GenCode1 : T_GenCode1 = ∇x:⟨nodes : [Node], Snodes : {Node : [Node]}⟩,
    y:⟨aT : Taddr, cseg : Tcseg, codeM : TcodeMap⟩.
var Acpt:num.var s:string.var n:Node.var v:num.
var l:[Node].var L:[Node].var n2:Node.
{
all :
x.nodes ♯ [?n]+?L ⇒
  x.nodes = x.nodes + L;   {
  g_decl :
    n ♯ ⟨class = %'decl', name =?s⟩ ⇒
      y.aT = y.aT + {s = Acpt}; Acpt = Acpt + 1
  g_aff :
    n ♯ ⟨class = %'aff', name =?s⟩ ⇒
      y.aT ♯ {s =?v} ⇒
        y.codeM = y.codeM + {n = [STORE, v]}
  g_op :
    n ♯ ⟨class = %'op', op =?s⟩ ⇒
      opMap ♯ {s =?v} ⇒
        y.codeM = y.codeM + {n = [v]}
  g_cmp :
    n ♯ ⟨class = %'cmp'⟩ ⇒
      y.codeM = y.codeM + {n = [CMP]}
  g_skip :
    n ♯ ⟨class = %'skip'⟩ ⇒
      y.codeM = y.codeM + {n = [NOP]}
  g_id :
    n ♯ ⟨class = %'id', name =?s⟩ ⇒
      y.aT ♯ {s =?v} ⇒
        y.codeM = y.codeM + {n = [GET, v]}
  g_cst :
    n ♯ ⟨class = %'cst', val =?v⟩ ⇒
      {
      r1 : y.csegI ♯ ? ⊞ {v =?v2} ⇒
        y.codeM = y.codeM + {n = [CGET, v2]},
      r2 : y.csegI = y.csegI + {v = len(y.cseg)}; y.cseg = y.cseg + {len(cseg) − 1 = v}
      }
  g_tst :
    n ♯ ⟨class = %'tst'⟩ ⇒
      y.Snodes ♯ ? + {n = [?, ?n2]} ⇒
        y.codeM ♯ ? + {n2 =?l} ⇒
          y.codeM = y.codeM + {n = [JE, +4, JR, len(l) + 2]}
  g_while :
    n ♯ ⟨class = %'while'⟩ ⇒
      y.Snodes ♯ ? + {n = [?, ?n2]} ⇒
        y.codeM ♯ ? + {n2 =?l} ⇒
          y.codeM = y.codeM + {n = [JE, +4, JR, len(l) + 4]}

  }
}
```

FIG. 9.1 : générateur de code, première passe

```
const GenCode2 := ∇x:⟨nodes : [Node], Snodes : {Node : [Node]}⟩,
      y:⟨codeM : TcodeMap, code : [num]⟩.
var n:Node.var n1:Node.var n2:Node.var n3:Node.
var l:[Node].var l1:[Node].var l2:[Node].var l3:[Node].
var L:[Node].
{
all :
x.nodes ♯ [?n]+?L ⇒
  x.nodes = x.nodes + L;
  {
  g_aff :
    n ♯ ⟨class = %'aff'⟩ ⇒
      x.Snode ♯ ?⊞{n = [?, ?n2]} ⇒
        y.codeM ♯ ?⊞{n2 =?l} ⇒
          y.code = y.code + l
  g_op :
    n ♯ ⟨class = %'op'⟩ ⇒
      x.Snode ♯ ?⊞{n = [?n1, ?n2]} ⇒
        y.codeM ♯ ?⊞{n =?l, n1 =?l1, n2 =?l2} ⇒
          y.code = y.code + l1 + l2 + l
  g_cmp :
    n ♯ ⟨class = %'cmp'⟩ ⇒
      x.Snode ♯ ?⊞{n = [?n1, ?n2, ?n3]} ⇒
        y.codeM ♯ ?⊞{n =?l, n1 =?l1, n2 =?l2} ⇒
          y.code = y.code + l1 + l2 + l
  g_tst :
    n ♯ ⟨class = %'cmp'⟩ ⇒
      x.Snode ♯ ?⊞{n = [?n1, ?n2, ?n3]} ⇒
        y.codeM ♯ ?⊞{n =?l, n1 =?l1, n2 =?l2, n3 =?l3} ⇒
          y.code = y.code + l1 + l + l2 + l3
  g_while :
    n ♯ ⟨class = %'while'⟩ ⇒
      x.Snode ♯ ?⊞{n = [?n1, ?n2]} ⇒
        y.codeM ♯ ?⊞{n =?l, n1 =?l1, n2 =?l2} ⇒
          y.code = y.code + l1 + l + l2 + [JMP, len(code)]
  }
}
```

FIG. 9.2 : générateur de code, seconde passe

Le langage *While* décrit en *Circus* : version visuelle et interactive

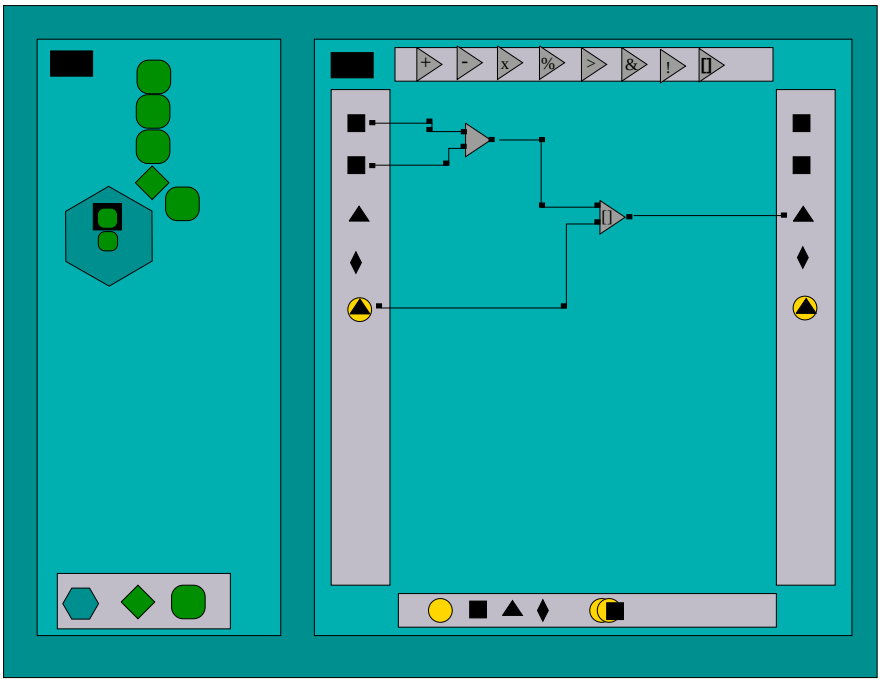


FIG. 10.1 : Une vue d'ensemble de *While* visuel

10.1 Présentation d'ensemble

Le langage visuel que nous présentons dans ce chapitre permet de programmer à deux niveaux d'abstraction différents, qui sont cependant en étroite relation :

1. *niveau structurel*. Le flot de contrôle est spécifié en juxtaposant des formes simples qui dénotent visuellement les instructions de base : affectations (carré), tests (losanges) et boucles (hexagones). Les propriétés de cette représentation sont la concision, la présentation synthétique de la structure du programme, l'accès rapide aux sous-ensembles (portions de code).
2. *niveau flot de donné*. Il correspond aux opérations complexes réalisées en partie droite des instructions d'affectation ou de test. Ces dernières ne sont rien d'autre que des transformations orientées de type entrée-sortie, connectées en cascade.

Le figure 10.1 montre une vue générale d'un programme *While* visuel : le contexte de gauche autorise la programmation au niveau structurel, et le contexte de droite permet d'exprimer les calculs.

10.1.1 Spécification de la structure du programme

Dans le contexte gauche, le programmeur utilise des opérations de type "dupliquer", "copier", "coller", "couper", "ajuster" sur les icônes situées dans le bas, afin de construire le graphe de contrôle de manière positionnelle. Le flot de contrôle s'écoule du haut vers le bas, et les losanges le font dévier vers la gauche (partie "then") ou vers la droite (partie "else"). Les hexagones sont spatialement interprétés de la même manière que les carrés. Le programmeur sélectionne une icône de la structure de contrôle avec le pointeur souris et une pression sur le bouton 1 afin de pouvoir travailler dans le contexte droit : en quelque sorte, pour pouvoir "remplir" l'élément graphique, ou encore lui donner une sémantique d'action. Selon

le type d'icône selectionné, le contexte droit est différent. Un carré (affectation), fait apparaître l'ensemble des variables dans la colonne de droite et la colonne de gauche. Sélectionner un losange ou un hexagone fait apparaître l'ensemble des variables dans la colonne de gauche, et une unique variable booléenne dans la colonne de droite (résultat de l'évaluation du test). La sous-section suivante détaille l'utilisation du contexte droit. L'hexagone, qui symbolise la boucle 'while' est différent des autres en ce sens qu'il peut contenir d'autres icônes, comme une boucle contient une séquence d'instructions. Des facilités de redimensionnement dynamique permettent de gérer l'espace graphique sans problème lié au facteur d'échelle.

10.1.2 Spécification des opérations de transformation

Dans le contexte droit, le programmeur trouve sur une palette verticale, à gauche et à droite, l'ensemble des variables déclarées dans le programme. Celle-ci sont visualisées par des assemblages de polygones, métaphores des assemblages de types : les polygones carrés sont des variables numériques, les triangles sont des chaînes, les losanges des booléens, et les cercles des tableaux. Il est possible de définir de nouvelles variables en sélectionnant un type dans la palette située horizontalement, dans le bas du contexte, et en le "tirant" dans la palette des variables. Dès lors, il n'est pas nécéssaire de nommer la variable : sa position suffit à la caractériser complètement. Cependant, le système calcule automatiquement un nom lors de l'opération, de façon à être compatible avec la version textuelle (l'arbre abstrait est la représentation commune). Cette possibilité est également une ouverture vers la conversion de programmes textuels en visuels, et vice-versa. Les constantes sont définies de manière identique, mais sont valuées lors de leur création (fenêtre de dialogue). Les types complexes (tableaux) peuvent être spécifiés directement dans la palette du bas, en superposant de gauche à droite des cercles (tableau de tableau de ...) et en plaçant le dernier élément (type terminal : numérique, booléen ou chaîne) dans le dernier cercle. Ils peuvent ensuite être utilisés pour définir des variables, comme les autres types primitifs. Le flot de transformation lui-même est spécifié en connectant les variables de gauche aux variables de droites au travers d'une "circuiterie" composée d'opérateurs. Chacun d'eux est orienté de la gauche vers la droite, porte un symbole qui explicite sa nature sémantique ("+", "-"...) et des ports de connection. Ici, la métaphore gauche-droite correspond à la sémantique classique de l'écoulement du temps.

10.2 Syntaxe de l'interaction

10.2.1 Spécification de la structure du programme

La grammaire de la figure 10.3 spécifie l'interaction dans le contexte gauche. Les lexèmes '+cg' et '-cg' correspondent à l'entrée/sortie du pointeur de souris dans le contexte gauche (carré gris avec type lexical 'cg'). La palette d'icônes est identifiée par le type lexical 'cgpal'. La "sous-grammaire" $moveEds$ illustre bien la puissance des grammaires LR(1) appliquées à la gestion d'événements : il est possible de pointer une icône, de presser le bouton 1 sans le relâcher, puis de déplacer le pointeur (l'icône doit suivre le déplacement, car aucun événement de sortie n'est émis : c'est donc une opération de "drag and drop" qui est attendue) et enfin, de relâcher la pression. L'icône doit alors rester à sa nouvelle position. Durant cette interaction, on ne peut entrer sur une icône autre qu'un hexagone (sinon, une erreur est signalée immédiatement : l'opérateur peut corriger et poursuivre son action). L'emboîtement d'icônes dans les hexagones est un cas difficile, car il doit être possible de changer de niveau d'emboîtement lors de l'interaction : par exemple l'utilisateur peut vouloir déplacer un carré situé dans un hexagone vers un autre hexagone, lui-même emboité dans un autre hexagone (configuration autorisée par la syntaxe visuelle :

const $ZERO : Pos = \langle x = 0, y = 0 \rangle$;
const $A_1 : \lambda$ **id** $Glyph$.[GET_MOUSE, PUSH, id, BUILD_SET, 1, PUSH, $ZERO$, SET_CSTR_POS]
const $ReacMove := \nabla x : \top, y : \langle bstack = [\top] \rangle$.
{
var $id : Glyph$;
$drag$:
  $y.bstack =? + [\langle cat =$ '+cg'$\rangle] +? + [\langle cat =$ '+square', $gid =?id \rangle, \langle cat =$ 'press_1'$\rangle]$
  $\Rightarrow$ $\langle$**put** $\langle typ =$ 'vop', $seq = A_1(id) \rangle$: VAMop
$drop$:
  $y.bstack =? + [\langle cat =$ '+cg'$\rangle] +? + [\langle cat =$ '+square', $gid =?id \rangle, \langle cat =$ 'press_1'$\rangle] +? + [\langle cat =$ 'release_1'$\rangle]$
  $\Rightarrow$ **put** $\langle typ =$ 'vop', $seq =$ [GET_MOUSE, UNSET_CSTR_POS]$\rangle$: VAMop
}

FIG. 10.2 : Un exemple de réaction sur action complexe : “drag and drop” emboité

correspond à la possibilité d'emboiter des boucles “while”). Les deux règles récursives $\langle moveCG \rangle$ et $\langle mCG \rangle$ capturent tout à fait naturellement cette difficile syntaxe d'action. Notons que la grammaire suppose l'existence de réactions graphiques en parallèle de l'analyse (celle qui permet à l'icône selectionnée de suivre le pointeur tant que la pression n'est pas relâchée). Ce type de réaction est spécifié par la machine abstraite de la figure 10.2.

10.2.2 Spécification des opérations de transformation

Nous nous concentrerons sur les interactions liées aux opérateurs de transformation, dans le contexte de droite (cf fig. 10.1). Ici, l'utilisateur sélectionne un opérateur dans la palette du haut, le déplace au centre du contexte. Ensuite, il sélectionne les ports d'entrées un à un pour les connecter aux opérandes de la palette gauche (variables). Enfin, il connecte le port de sortie de l'opérateur, soit à une variable de la palette droite (elles peuvent être déja utilisées en entrée, comme dans $a = a + 1$), soit au port d'entrée d'un autre opérateur. Lorsque l'édition est terminée, le programmeur peut lancer la compilation (incrémentale) de sa spécification : elle consiste à générer l'arbre abstrait correspondant, et à contrôler les types (la génération est faite au niveau global). Pour cela, il doit cliquer sur le bouton situé dans l'angle supérieur gauche. Les erreurs éventuelles apparaissent sous forme d'icônes colorées (triangles pointe en bas), rouges pour une erreur, roses pour une alerte. Ces informations restent présentes tant que l'erreur n'est pas corrigée, même si le programmeur change la sélection du bloc dans la représentation structurelle du contexte gauche. Ainsi, à tout moment, l'utilisateur garde sous les yeux l'ensemble des erreurs qui lui restent à traiter dans la globalité du programme. De plus, en sélectionnant une des icônes d'erreur, une animation est lancée qui spatialise l'origine de l'erreur dans le contexte droit, et un compte rendu textuel est affiché (voir [118] pour un autre exemple d'utilisation de cette technique). Lorsque la compilation est réussie, le bloc du contexte gauche change de couleur, afin d'offrir au programmeur une vision globale de l'avancement de son travail. La grammaire d'action ne présentant pas de différence fondamentale avec la précédente, elle ne sera pas produite dans le détail.

10.3 Syntaxe visuelle

10.3.1 représentation structurelle

L'analyse parcourt les polygones du haut vers le bas, pour les carrés et les hexagones. Ces derniers sont de plus explorés “en profondeur” : ils doivent également contenir une phrase visuelle (un hexagone

```
grammar Basics                {
⟨click1⟩                →     'press_1'  'release_1';
⟨click2⟩                →     'press_2'  'release_2';
}

grammar createEdS       :     Basics{
⟨createCG⟩              →     '+cgpal'  ⟨doCGpal⟩  '-cgpal'⟨click1⟩        //entrée/sortie dans la palette
                              //suivie d'une pression bouton, dans le contexte
                              |  '+cgpal'  '-cgpal'                        //pas d'action ds la palette
                              ;
⟨doCGpal⟩               →     '+square'  ⟨click1⟩  '-square'                    //sel. affectation
                              |  '+losange'  ⟨click1⟩  '-losange'                //sel. test
                              |  '+hexagone'  ⟨click1⟩  '-hexagone'              //sel. boucle
                              |  '+hexagone'  ⟨doCGpal⟩  '-hexagone'             //sel. boucle imbriquée
                              ;
}
grammar selectEdS       :     Basics{
⟨selectCG⟩              →     '+square'  ⟨click2⟩  '-square'                    //sel. affectation
                              |  '+losange'  ⟨click2⟩  '-losange'                //sel. test
                              |  '+hexagone'  ⟨click2⟩  '-hexagone'              //sel. boucle
                              |  '+hexagone'  ⟨selectCG⟩  '-hexagone'            //sel. à l'intérieur d'une boucle
                              ;
}

grammar moveEdS         :     Basics{
⟨moveCG⟩                →     '+square'  'press_1'  ⟨mCG⟩  '-square'            //sel. affectation
                              |  '+losange'  'press_1'  ⟨mCG⟩  '-losange'        //sel. test
                              |  '+hexagone'  'press_1'  ⟨mCG⟩  '-hexagone'      //sel. boucle
                              |  '+hexagone'  ⟨moveCG⟩  '-hexagone'             //sel. à l'intérieur d'une boucle
                              ;
⟨mCG⟩                   →     'release_1'
                              |  '+hegagone'⟨mCG⟩  '-hegagone'
                              ;
}

grammar EdS             :     Basics{
⟨CG⟩                    →     '+cg'  ⟨doCG⟩  '-cg'                          //entrée/sortie dans le contexte
⟨doCG⟩                  →     ⟨doitCG⟩  ⟨doCG⟩                              //repétition des sous-actions
                              |  ⟨doitCG⟩
                              ;
⟨doitCG⟩                →     ⟨createCG⟩  |  ⟨selectCG⟩  |  ⟨moveCG⟩       //les trois grandes classes d'actions
                              ;
}
```

FIG. 10.3 : La grammaire d'action pour l'édition structurelle du programme

Rgrammar AV1 {
terms : $square : \langle cat :$ **string in**{'square'}$, val : G\_square\rangle$;
$losange : \langle cat :$ **string in**{'losange'}$, val : G\_losange\rangle$;
$hexagone : \langle cat :$ **string in**{'hexagone'}$, G\_hexagone\rangle$;
nterms : $A : \langle cat :$ **string in**{'A'}$, val : S\_Glyph\rangle$;
$B : \langle cat :$ **string in**{'B'}$, val : S\_Glyph\rangle$; $S : \langle cat :$ **string in**{'S'}$\rangle$;
rules :

$a : S \rightarrow b : A$;

$a : A \rightarrow b : B$ [Above$(b.val, c.val)$ **and** Near$(b.val, c.val)$] $c : A$
$\{a.val = c.val\}$
| $b : B$
$\{a.val = b.val\}$
;

$a : B \rightarrow b : square$
$\{a.val = b.val\}$
| $b : hexagon$ [Contains$(b.val, c.val)$] $c : A$
$\{a.val = b.val\}$
| $b : losange$
[Above$(b.val, c.val)$ **and** Near$(c.val, b.val)$ **and** OnRight$(b.val, c.val)$]
$c : A$
[Above$(b.val, d.val)$ **and** Near$(c.val, b.val)$ **and** OnLeft$(b.val, d.val)$]
$d : A$
$\{a.val = b.val\}$
;

}

FIG. 10.4 : La grammaire *PLR* générant l'analyseur syntaxique visuel (contexte gauche)

vide produit une erreur de syntaxe). La figure 10.4 présente la grammaire analysant le contexte gauche. La relation spatiale d'emboîtement des hexagones impose de propager les informations de taille (attribut *size*) dans les non-terminaux A et B. Notons également que cette syntaxe laisse une certaine liberté dans les positions relatives des icônes (*Above, Onright* et *OnLeft* ne sont que des inégalités). Toutefois, le prédicat *Near* permet de lever les ambiguités de l'analyse tout en laissant une certaine souplesse.

10.3.2 Spécification des opérations de transformation

Ici, l'analyse progresse spatialement de la droite vers la gauche. Elle analyse les interconnections entre variables et opérateurs, réalisées au moyen de segments de droites. La figure 10.5 montre la grammaire positionnelle correspondant à la syntaxe des interconnections. Seuls, les opérateurs unaires et binaires ont été considérés (des opérateurs ternaires peuvent être conçus sur le même principe). La règle liée au non-terminal $varL$ débute l'analyse en "cherchant" une variable située dans la colonne de droite et connectée à un élément graphique situé à sa gauche. Cette règle utilise la position relative de la colonne (matérialisée par un rectangle vertical), à l'intérieur du contexte droit (grand carré).

10.4 Couplage avec les noyaux de traitement

Nous envisageons ici le contrôle des types et la génération de code. Dans les deux cas, l'arbre de syntaxe abstraite sert de structure "pivot". L'analyse syntaxique visuelle élabore un (sous-)arbre qui doit être transformé en un (sous-)arbre syntaxique abstrait, par une machine comparable aux transducteurs proposés pour *While* textuel. Nous avions proposé un contrôleur de type capable de travailler sur

Rgrammar AV2 {
terms :
$square : \langle cat : \textbf{string in}\{\text{'square'}\}, val : G\_square\rangle;$
$line : \langle cat : \textbf{string in}\{\text{'line'}\}, val : G\_Line\rangle;$
$port : \langle cat : \textbf{string in}\{\text{'port'}\}, val : G\_SmallBlackSquare\rangle;$
$losange : \langle cat : \textbf{string in}\{\text{'losange'}\}, val : G\_Losange\rangle;$
$triangle : \langle cat : \textbf{string in}\{\text{'triangle'}\}, val : G\_Triangle\rangle;$
$rectangle : \langle cat : \textbf{string in}\{\text{'rectangle'}\}, val : G\_Rectangle\rangle;$
$op1 : \langle cat : \textbf{string in}\{\text{'op1'}\}, val : G\_TriangleL\rangle;$
$op2 : \langle cat : \textbf{string in}\{\text{'op2'}\}, val : G\_TriangleL2\rangle;$
nterms :
$A : \langle cat : \textbf{string in}\{\text{'A'}\}, val : Pos\rangle;$
$B : \langle cat : \textbf{string in}\{\text{'B'}\}, val : Pos\rangle;$
$S : \langle cat : \textbf{string in}\{\text{'S'}\}, val : Pos\rangle;$
$ctxR : \langle cat : \textbf{string in}\{\text{'ctxR'}\}, val : Pos\rangle;$
$link : \langle cat : \textbf{string in}\{\text{'link'}\}, From : Pos, To : Pos\rangle;$
$var : \langle cat : \textbf{string in}\{\text{'var'}\}, val : Pos\rangle;$
$varR : \langle cat : \textbf{string in}\{\text{'varR'}\}, val : Pos\rangle;$
$varL : \langle cat : \textbf{string in}\{\text{'varL'}\}, val : Pos\rangle;$
rules :

$a : S \rightarrow b : A \quad \{a.val = b.val\};$

$a : A \rightarrow b : varL \quad [Near(b.val, c.From)\ \textbf{and}\ OnLeft(c.From, b.val)]$
$c : link \quad [Near(c.To, d.val)] \quad d : C \quad \{a.val = d.val\};$

$a : B \rightarrow b : op1 \quad [Near(b.val, c.From)\ \textbf{and}\ OnLeft(c.From, b.val)]$
$c : link \quad [Near(d.val, c.From)\ \textbf{and}\ OnRight(c.From, b.val)]$
$d : C$
$\{a.val = b.val\}$
$\mid\ b : op2$
$[Near(b.val, c.From)\ \textbf{and}\ OnLeft(c.From, b.val)\ \textbf{and}\ Above(c.From, b.val)]$
$c : link$
$[Near(c.To, d.val)\ \textbf{and}\ OnRight(c.From, d.val)]$
$d : C$
$[Near(b.val, e.From)\ \textbf{and}\ OnLeft(e.From, b.val)\ \textbf{and}\ Above(b.val, e.From)]$
$e : link$
$[Near(e.To, f.val)\ \textbf{and}\ OnRight(e.From, f.val)]$
$f : C$
$\{a.val = b.val\};$

$a : C \rightarrow b : varR \quad \{a.val = b.val\} \quad \mid \quad b : B \quad \{a.val = b.val\};$

$a : link \rightarrow b : port \quad [\text{ToPos}(c) == b.pos] \quad c : line \quad [\text{FromPos}(c) == d.pos] \quad d : link$
$\{a.To = d.To, a.From = b.val\}$
$\mid\ b : port \quad [\text{ToPos}(c) == b.pos] \quad c : line \quad [\text{FromPos}(c) == d.pos] \quad d : port$
$\{a.To = d.val, a.From = b.val\};$

$a : varL \rightarrow b : ctxR \quad [Contain(b.val, c.val)\ \textbf{and}\ OnLeft(c.val, b.val)]$
$c : rectangle \quad [Contain(c.val, d.val)] \quad d : var \quad \{a.val = d.val\};$

$a : varR \rightarrow b : ctxR \quad [Contain(b.val, c.val)\ \textbf{and}\ OnLeft(b.val, c.val)]$
$c : rectangle \quad [Contain(c.val, d.val)] \quad d : var \quad \{a.val = d.val\};$

$a : var \rightarrow b : varT\{a.val = b.val\}$
$\mid\ b : array \quad [Contain(b.val, c.val)] \quad c : varT \quad \{a.val = c.val\};$

$a : array \rightarrow b : circle \quad \{a.val = b.val\}$
$\mid\ b : circle \quad [Overlapp(b.val, c.val)\ \textbf{and}\ OnLeft(b.val, c.val)] \quad c : array$
$\{a.val = c.val\};$

$a : varT \rightarrow b : square \quad \{a.val = b.val\}$
$\mid\ b : triangle \quad \{a.val = b.val\} \quad \mid \quad b : losange \quad \{a.val = b.val\};$

}

FIG. 10.5 : La grammaire *PLR* générant l'analyseur syntaxique visuel (contexte droit)

des sous-arbres, à la reception de messages déposés dans la mémoire de coordination (cf $ProcTypes$, **??**). Il est donc possible à tout moment de réaliser le contrôle de type sur des expressions partiellement incomplètes, dès lors que toutes les feuilles du sous-arbre sont définies dans un contexte cohérent. Les erreurs sont transmises par l'intermédiaire de la mémoire de coordination, puis interpretées dans le contexte visuel de l'analyse.

10.4.1 contrôle des types

Nous présentons trois stratégies pouvant être implantées simultanément ou en proportions variables, selon la nature du langage visuel. Dans *while,* ces trois niveaux sont présents d'une manière plutôt équilibrée.

1. retour visuel direct. Les éléments visuels sont suffisamment expressifs pour prévenir les erreurs de types : l'information relative au typage des variables est en permanence "sous les yeux", dans la mémoire perceptuelle du programmeur. Elle ne nécessite donc pas de traitements cognitifs particuliers, ni d'efforts de mémorisation. Ici, c'est la forme des variables qui exprime leur type. Cette information est donc constamment associée aux manipulations sur les variables, et de ce fait susceptible d'abaisser le taux d'erreur de manière significative, à la source même de la spécification.
2. retour visuel interactif. A ce niveau, les informations visuelles sont présentées aux différentes phases de l'interaction, de manière dynamique : si le programmeur sélectionne d'abord un opérateur, le système peut orienter le choix des opérandes en cachant les variables dont les types sont incompatibles avec l'opérateur. Une autre solution consiste à montrer à coté du pointeur, les formes (donc les types) compatibles. Dans l'idéal, la collaboration "homme-machine" s'adapte à toutes les possibilités d' interaction : si l'utilisateur sélectionne d'abord une opérande d'entrée, le système peut alors proposer les opérateurs compatibles (en arité et type). Si c'est la variable de sortie qui est d'abord choisie, alors les opérateurs doivent être proposés en tenant compte des fonctions de transfert de type. Idéalement il devrait être possible de tenir compte du contexte pour orienter les choix du programmeur : dans le cas d'une transformation complexe faisant intervenir plusieurs opérateurs en cascade, les variables intermédiaires peuvent être intégrées automatiquement dans les choix de sélections proposés au programmeur.
3. retour visuel différé. Lorsque les niveaux précédents ne suffisent pas à assurer l'ensemble des vérifications, il est nécessaire de réaliser un traitement plus "profond", sur demande de l'opérateur (selection d'un bouton dédié). Dans ce cas, les erreurs potentielles doivent être richement reliées au contexte d'édition. Dans *While* visuel, la possibilité de cliquer sur les icônes d'erreur pour localiser les éléments incriminés est un exemple d'application de cette stratégie.

10.4.2 génération

Dans le langage visuel que nous proposons comme support d'étude, la génération de code est réalisée de manière globale, sur demande, lorsque toutes les unités structurelles du contexte gauche sont spécifiées sans erreurs. Toutefois, il serait facile d'envisager une génération plus incrémentale, associée à la vérification des types dans le contexte droit. Il faudrait par exemple changer la signature de la machine $GenCode1$ (cf figure 9.1) qui réalise la première passe (calcul des adresses et des fragments de code), de manière à traiter incrémentalement le segment de code et la table des fragments. Cette modification est réalisable par une composition simple ($\otimes$) avec une machine ayant la signature requise. Il reste alors à séparer les deux passes : dans la version textuelle de *While*, elles sont enchaînées de manière séquentielle. A présent, la première passe doit être appliquée sur chaque expression arithmétique, lorsque la vérification de type

est réussie. L'environnement *Circus* offre les facilités compositionnelles en rapport avec ce type de comportement. Notons encore que les structures intermédiaires utilisées pour la première passe permettent les redéfinitions "locales" à condition de prévoir la destruction des ressources associées. Une machine dédiée doit être capable de calculer les différentiels induits par les redéfinitions afin de libérer les entrées devenues inutiles, dans les tablaux associatifs $cseg$ et $codeM$.

10.5 Synthèse

Cette partie présente la mise en œuvre de l'environnement *Circus*, aussi bien dans le traitement des langages textuels que visuels. Le système de type proposé dans la partie théorique de l'étude, ainsi que les possibilités compositionnelles des machines abstraites polymorphes ont été mis à profit pour concevoir l'architecture d'un compilateur expérimental d'une complexité suffisante pour être représentative. Plusieurs machines ont été proposées en illustration de l'expressivité de notre langage et de sa capacité à résoudre les problèmes de transformations complexes dans lesquels des aspects temporels de synchronisation et de parallèlisme peuvent être pris en compte.

Les langages visuels sont abordés dans un cadre original, basé sur un modèle de représentation possédant une structure triple :

1. Unités visuelles structurées et polymorphes. Ce sont les "mots" de base, ou lexèmes visuels, appelés *glyphes*, constituant le vocabulaire de base de tout langage visuel envisagé dans le cadre de cette étude. Leur structure est conçue de manière à simplifier les concepts requis pour la spécification des opérations visuelles, mais également pour augmenter la puissance des traitements visuels (transformations multi-dimensionnelles, spécification des formes) ainsi que leur capacité à transmettre de l'information.
2. Interactions structurées. Basées sur une sémantique d'événements très simple, des grammaires "classiques" associées à un analyseur syntaxique spécialisé pour le traitement de flots continus et temps réel permettent de rendre explicite les lois de l'interaction et de réaliser un premier niveau de vérification syntaxique. De plus des actions graphiques peuvent être associées aux phases de reconnaissance, afin d'offrir des possibilités interactives accrues, indispensables à la réalisation de langages visuels dynamiques, les plus prometteurs.
3. Syntaxe visuelle. Les assemblages d'objets graphiques obeissent à une syntaxe spatiale précise, rendue explicite au moyen de grammaires positionnelles et d'analyseurs associés.

Ces abstractions du modèle de représentation sont implantées au moyen d'une machine visuelle exécutant un code "à pile" afin de lui donner une sémantique perceptuelle. Cette machine est également en charge de gérer les informations nécessaires à la génération des événements d'interaction et à leur liaison avec le contexte visuel. Toutefois, aucun langage de plus haut niveau n'a été proposé pour spécifier les opérations graphiques. Ainsi la programmation de la machine visuelle se fait au niveau d'un "langage machine" spécialisé. De même, si les grammaires d'action et les grammaires visuelles sont à l'évidence dépendantes et complémentaires, aucun lien formel n'a été établi par cette étude, bien que de nombreuses possibilités soient ouvertes.

Une version textuelle et visuelle du même langage a été présentée dans le but de démontrer l'unité profonde apportée par l'environnement *Circus*, ainsi que la réutilisation effective de certains noyaux de traitement. Bien que les possibilités de réécriture de graphes n'aient étés que succinctement explorées (essentiellement pour des raisons technique de présentation), la présence de structures de données riches et de moyens de transformation adaptés sont les deux clés de notre approche visant à jeter un pont entre les technologies des langages textuels et visuels.

Conclusion

11.1 Récapitulatif

La première partie de ce mémoire s'est attachée à placer la problématique de la construction des langages de programmation passés, actuels et futurs ainsi que leurs environnements de développement dans un cadre commun, la transformation de structures. Nous avons alors identifié les caractéristiques importantes d'un générateur pour l'aide à la construction d'environnements orientés langage : compositionnalité, réutilisabilité, niveaux d'abstraction multiples (haut niveau : grammaires et autres spécifications, niveau intermédiaire : langage spécialisé dans la transformation) et richesse de la modélisation des structures. Ces qualités étant naturellement justifiées par l'analyse de la première partie, et venant en réponse aux problèmes génériques liés à la conception et l'implantation d'environnements de développement élaborés.

La seconde partie c'est alors concentrée sur la définition formelle du langage de niveau intermédiaire *Circus* , en s'attachant plus particulièrement au système de types, ce dernier étant vu comme l'élément intégrateur des différents concepts, parfois *a priori* antagonistes, que nous avons voulu réunir. La définition formelle du langage s'est accompagnée de la caractérisation des propriétés les plus importantes, à savoir la correction opérationnelle du système de type (pour le noyau fonctionnel) et l'existence d'algorithmes autorisant le contrôle effectif des types.

La troisième partie s'est proposée d'illustrer la mise en œuvre de *Circus* en esquissant la réalisation d'un environnement de développement textuel et visuel pour le mini langage d'étude *While*. Cette description s'est fixée pour but de montrer comment les différentes qualités identifiées dans la première partie sont produites par les solutions que nous avons proposées, mais également leur bien fondé, en ce sens qu'elles permettent effectivement de simplifier la conception et la réalisation des outils de traitement de langage. D'autre part, nous espérons que le lecteur sera convaincu avec nous qu'une telle approche permet de franchir une barrière qualitative, en ce sens que les performances fonctionnelles de ces outils se situent potentiellement bien au dessus de l'existant.

11.2 Étude du langage

L'étude du langage s'est concentrée sur la *formalisation* mathématique des concepts, afin de leur donner un contenu précis. Toutefois, certains aspects importants n'ont pas été abordés, comme les algorithmes de filtrage performants nécessaires à une implantation efficace et réaliste des machines abstraites polymorphes. D'autres aspects, liés au système de type ainsi qu'au modèle de coordination et de concurrence, ont été considérés comme plus difficiles à traiter en vue de l'intégration dans un cadre de réécriture généralisée à des structures de données variées. De nombreuses propriétés mathématiques essentielles devraient encore être établies afin de conférer une complète assise théorique au travail proposé ; en effet, si le noyau du langage est fortement caractérisé, il n'en est pas de même pour les extensions visant à enrichir les types de données, le contrôle de l'exécution et les algorithmes de contrôle des types. De même, il serait intéressant de produire une sémantique dénotationnelle des types, plus "pure" d'un point de vue mathématique, plus apte à mettre en évidence les propriétés struturelles des types de données proposés.

Ce travail théorique a probablement souffert de ce qui à notre sens lui confère également ses qualités : il a été guidé par des intuitions computationnelles, issues d'une approche expérimentale et pragmatique. La difficulté a donc été de capturer ces intuitions, de les contraindre à la formalisation mathématique. Il serait toutefois faux de dire que le travail formel n'a pas lui-même influencé la perception pragmatique. Bien au contraire, le travail de formalisation a permis de structurer des concepts parfois confus ou redondants, de simplifier la forme du langage en la ramenant à l'essentiel, mais surtout de révéler les nombreux

points de problèmes liés aux intuitions d'origine. En ce sens, on peut dire que l'étude théorique a rempli son rôle d'une manière fertile.

Il reste donc à rappeler les concepts les plus originaux issus de cette synthèse :

1. *Système de types.* Il offre une relation de sous-typage basée essentiellement sur l'inclusion ensembliste. Deux opérateurs sur les types, $\oplus$ et $\otimes$ permettent d'exploiter finement les informations liées aux domaines de valuation, et de les propager dans des termes reflétant un comportement à l'exécution pouvant dépendre des valeurs, comme le test **if then else** , l'itération $*()$ ou le filtrage. Ainsi, il devient possible de faire apparaître dans le traitement des types des propriétés opérationnelles, parfois abordées à un autre niveau d'analyse.
2. *Contrôle d'exécution. Circus* propose une synthèse des trois grandes tendances usuellement rencontrées dans les langages de programmation : le *fonctionnel*, l'*impératif* et le *déclaratif*. Bien sur, les λ fonctions ne sont pas récursives (mais aucun obstacle de fond n'empêche de les proposer), les instructions impératives sont réduites à l'essentiel (affectation,itération,test), et les règles sont assujetties à un schéma d'exécution explicite. Mais toutefois, nous considérons que l'essentiel de chacun de ces concepts trouve dans *Circus* une synthèse efficace, unifié par le système de types. L'expérimentation nous montrera l'impact réel de cet équilibre dans les paradigmes de programmation.
3. *Filtrage et règles.* C'est ce double concept qui donne à *Circus* sa vocation de langage spécialisé dans la transformation de structures. En effet, on trouve réunies en une seule opération trois phases omniprésentes lorsque l'on doit traduire une structure source en une structure cible (la réécriture pouvant être considérée comme un cas particulier) : (i) tests portant sur la forme de la structure source, (ii) extraction de certaines informations localisées au sein de cette même structure et (iii) création ou décoration de la structure cible en utilisant les informations de (i) et (ii). Nous sommes dans le cas typique où l'abstraction de contrôle s'accorde exactement au paradigme du langage. Une seule règle de filtrage complexe peut remplacer plusieurs dizaines d'instructions conventionelles, sans nécessairement accuser de lourdes pertes de performance.
4. *Composition de machines abstraites polymorphes.* Pourquoi ne pas additionner ou multiplier des objets, des fonctions, afin de produire de nouvelles entités, capables de produire de nouveaux comportements intéressants ? Essentiellement parce qu'il est difficile de définir une sémantique pertinente pour ces opérations. C'est pourtant le défi relevé par les modèles de programmation orientés objets, où la composition se fait par héritage et surcharge. Au-delà du progrés que ces mécanismes ont apporté à l'ingénierie du logiciel, ce modèle de composition est fortement complexifié par les subtilités liées aux hypothèses d'exécution (comportement des constructeurs et destructeurs, disponibilité de l'autoréférence, et ambiguités dans le graphe d'héritage). D'autre part, la sémantique de composition par héritage et surcharge, essentiellement l'ajout et la substitution de méthodes au pool apporté par la hiérarchie, reste pauvre. On souhaiterait pouvoir bénéficier de connecteurs plus riches, permettant de manipuler "algébriquement" les abstractions de contrôle, afin de mieux tirer partie des connaissances qu'elles encapsulent, mais également afin de pouvoir les recomposer plus richement. *Circus* réalise un pas dans cette direction en prenant en compte la sémantique de la composition, mais aussi l'impact sur le système de types. En effet, il est important de conserver la "protection" apportée par le contrôle des types, qui préserve l'exécution de bon nombre d'erreurs pouvant être engendrées par les opérations de composition elles-mêmes.
5. *Machine abstraite visuelle.* La notion de machine abstraite est nouvelle dans le domaine des langages visuels, bien que très connue en théorie et pratique des langages de programmation textuels. Dans le cadre de notre étude, elle permet d'offrir une abstraction simplificatrice des principales

fonctions de manipulation des objets visuels. De plus, elle offre une solution efficace pour intégrer les traitements visuels aux traitements transformationnels réalisés par les machines abstraites polymorphes. En effet, le programmeur exploite la machine visuelle au travers des mécanismes de coordination offerts en standard pour toutes les machines abstraites de *Circus* .

11.3 Expérimentation

Les différents concepts explorés lors de ce travail scientifique n'ont pas été expérimentés globalement. Toutefois, cette étude s'appuie sur des connaissances concrètes acquises au fil d'expérimentations partielles et orientées sur des aspects précis, dans le cadre de compilateurs et outils de traitements réalistes. Ainsi le modèle transformationnel a été utilisé pour développer des compilateurs (textuels) performants pour les langages *Olan* et *CLF*. Les générateurs d'analyseurs syntaxiques (utilisant l'héritage de grammaires) et lexicaux sont réalisés, mais produisent dans l'état actuel des structures de plus bas niveau que *Circus* . Le modèle visuel, la machine visuelle et l'analyse syntaxique des interactions a été concrètement explorée et validée au travers d'un prototype de langage visuel de configuration pour applications *CLF* distribuées. Les résultats de ce travail ont été publiés, et sont joints en annexe. D'autre part, les algorithmes de contrôle de types ont été implantés et extensivement testés. La compilation des machines abstraites reste un point important en cours d'expérimentation. De nombreux éléments permettent selon nous de prédire de bonnes performances :

1. composition au niveau syntaxique. C'est le cas idéal en ce sens que l'optimisation est applicable en aval de la composition. Il est donc possible de déterminer et d'éliminer les tests redondants susceptibles d'être apportés par l'opération de composition elle-même.
2. typage fort. Il évite (partiellement) de réaliser des vérifications de type pendant l'exécution. Ou plutôt, il minimise les tests à l'exécution concernant les types, car le filtrage de certaines expressions nécessite tout de même de telles évaluations.

Cependant, des techniques d'optimisation, non triviales, doivent encore être étudiées, notamment pour éliminer les tests redondants dans les parties gauches de règles applicables séquentiellement ou alternativement. Cette approche prometteuse passe par la définition d'une forme normale pour les tests de filtrage, ce qui ne semble pas particulièrement difficile étant donné la nature du système de type.

11.4 Perspectives et enjeux

Le langage *Circus* fait aujourd'hui l'objet d'un projet de recherche à *XRCE*. Le compilateur est partiellement implanté, et sera amené à un niveau de réalisation suffisant pour une nouvelle campagne d'expérimentation. Cette dernière permettra de valider ou reconsidérer certain choix relatifs à l'expressivité. Elle permettra également d'identifier les domaines ou le langage peut être étendu, théoriquement et pratiquement (librairies, opérateurs prédéfinis, constructions syntaxiques, modèle). Un premier axe d'application d'un tel langage est la transformation de documents numériques structurés- automatique, semi-automatique ou interactive- où les structures logiques et graphiques sont mises en correspondance grâce à des traitements complexes s'apparentant aux processus de compilation. Cet axe est aujourd'hui abordé par le langage *XSLT*, de plus haut niveau que *Circus* si l'on considè l'expressivité, mais par la même plus restreint quant à la richesse des oérations. Les applications potentielles sont liées à la conversion de formats logiques (DTDs), de formats physiques et diverses opérations de "rendering". On peut y ajouter une classe de transformations plus difficiles et plus prometteuses liée au contenu du document :

transformations linguistiques, enrichissement par annotation automatique ou manuelle de méta-données. Le second axe concerne la création d'outils visuels interactifs dont la sémantique est fortement contrainte, et pour lesquels la visualisation de l'information est importante (au sens où l'espace d'information est par nature complexe mais se prêtant bien à la représentation visuelle). En restant dans le domaine du traitement documentaire, un tel outil pourrait prendre la forme d'un langage visuel permettant de spécifier le format physique d'un document structuré *XML*. Ce genre de spécification complexe repose sur une sémantique de style très riche (*DSSSL*) exprimée textuellement bien que fondamentalement visuelle. La grammaire spécifiant la forme logique des arbres représentant les documents (appelée *DTD* pour *Document Type Definition*), conjointement aux connaissances sur la sémantique de style pourrait guider l'interaction utilisateur via une architecture de composants *Circus* fonctionnant en collaboration avec la machine abstraite visuelle.

Outre les aspects applicatifs, l'étude théorique ouvre certaines voies qu'il serait intéressant d'explorer :

Enrichissement du système de types formel. La méthode suivie - extension progressive d'un noyau de langage - a nécessité une structure particulière de la syntaxe et du système de transition apportant modularité et extensibilité. Ces qualités laissent envisager l'extension du système de type : exceptions, références, types récursifs, types unions, polymorphisme paramétrique. Plus "exploratoire" : les types enumérés de Circus pourraient peut être se généraliser en types "intentionnels", où l'ensemble des valeurs dénotées par le type serait décrit par un prédicat logique, plutôt que extensivement. Cela suppose la décidabilité de la relation de conformité et de sous-typage, et donc un système d'inférence logique complet et décidable. Par exemple, un type énuméré tel que

$$\text{type } t = \textbf{num in}\{10, 11\}$$

pourrait s'écrire de manière équivalente par

$$\text{type } t = \{x : \textbf{num} \mid x = 10 \textbf{ or } x = 11\}$$

Le type "entiers naturels" et "entiers dans l'intervalle $[1, 10]$ par

$$\begin{array}{l}\textbf{type integer} = \{x : \textbf{num} \mid x \geq 0\}\\ \textbf{type int1to10} = \{x : \textbf{num} \mid x \geq 1 \textbf{ and } x \leq 10\}\end{array}$$

Dans ce cas, la relation de sous-typage (notée $\preccurlyeq$) devrait permettre d'affirmer

$$\text{int1to10} \preccurlyeq \text{integer}$$

et donc que

$$(x \geq 1 \textbf{ and } x \leq 10) \Rightarrow (x \geq 0)$$

Type graphe. L'intégration du type de données "graphes" dans *Circus* a fait l'objet de nombreux efforts, car ce type de donnée nous semblait particulièrement apte à modéliser les structures les plus complexes. Hélas, les résultats obtenus n'ont pu atteindre le niveau de clarté requis pour une exposition concise dans le cadre de ce travail. Les graphes conçus comme des types de données fortement intégrés au langage (syntaxe et sémantique) posent de nombreux problèmes. Un de ces derniers, rarement abordé ou évoqué, concerne l'expressivité, prise au sens "utilisateur" : quelle est la complexité par exemple d'une expression qui spécifie les opérations primitives telles que ajout d'un nouveau nœud et insertion d'arcs ? On

s'apperçoit qu'une approche textuelle produit très rapidement des phrases longues et absconses, voire même illisibles pour des spécifications plus riches telles que le filtrage. L'introduction des graphes demande donc un traitement particulierement soigné de la syntaxe, avec une orientation vers l'utilisateur, ce qui requiert un savoir faire assez mal connu. La plupart des auteurs travaillant dans ce domaine utilisent des formalismes visuels parfois peu rigoureux, qui esquivent ces difficultés fondamentales pour se concentrer sur les problèmes de sémantique. En effet, si il est absolument nécessaire que la sémantique des opérations portant sur les graphes soit claire et définie, il est tout aussi important que le programmeur perçoive facilement la nature de ces opérations et puisse les utiliser sans être découragé par leur formulation. Une étape importante dans la définition serait de proposer une syntaxe et sémantique claire des opérations de construction, concaténation et filtrage de graphes, afin de les intégrer au langage comme un type de donné particulièrement générique. Cet objectif requiert également la conception d'un support d'exécution performant offrant des opérations fondamentales telles que calcul de fermeture, fermeture transitive, projection et linéarisation.

Abstractions de haut niveau - Surlangages. Dans notre étude, nous avons implanté quelques abstractions de plus haut niveau que le langage intermédiaire : langage de description de grammaires, générateur d'analyseurs lexicaux et d'analyseurs syntaxiques LR(1). D'autres seraient intéressants, tels que des transducteurs hors-contextes, pour transformer des arbres syntaxiques concrets en arbres abstraits, ou des analyseurs syntaxiques plr(1) (analyse visuelle cf [38]). De même, un langage textuel pour spécifier des applications visuelles interactives pourrait probablement utiliser avec succès les abstractions visuelles et interactives offertes par la machine abstraite visuelle de *Circus*.

Le code graphique est aujourd'hui de très bas niveau : il est remarquable de pouvoir envisager un langage dont la sémantique est purement visuelle avec une syntaxe textuelle. Ainsi, une expression telle que

$$g.pos = (g_2.pos + g_3.pos)/2$$

pourrait être schématiquement traduite en

$$[PUSH, g, PUSH, g_2, GET\_POS, PUSH, g_3, GET\_POS, ADD\_PN, PUSH, 0.2, DIV\_PN, SET\_POS\_TO]$$

dont la sémantique visuelle est de positionner le glyphe g en coïncidence avec le milieu géométrique de g_1 et g_2. Dans le même esprit, les déclarations de types visuels pourraient faire l'objet de vérifications au regard de leur utilisation (un type de glyphe dont la couleur est jaune uniquement impose que ses instances conservent la couleur jaune tout au long de leur utilisation).

A l'issue de ce travail, *Circus* apparait comme un "assembleur" de haut-niveau pour le traitement des languages, ou bien encore une plateforme de bas-niveau offrant de nombreuses possibilités de composition et de traitements adaptés à des structures de données riches. A un niveau d'abstraction supérieur, les générateurs d'analyseurs utilisent les grammaires pour produire du code circus, réutilisable et composable. De nombreux autres outils seraient suceptibles de compléter l'aide à la réalisation de langage : générateurs de transducteurs de syntaxe concrète en syntaxe abstraite, analyseurs et optimiseurs de grammaires. Plus spécifiquement, un outil analysant et vérifiant les rapports entre grammaire d'action et analyse syntaxique visuelle présente un réel intérêt pratique. Sa base théorique pourrait s'appuyer sur des raisonnements en logique spatiale (cf [53]), à partir des cercles englobant et cercles inscrits associés aux glyphes.

Bibliographie

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM TRansaction on Programming Languages and Systems*, 13(2) :237–268, April 1991.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. Technical Report TR-120, Digital Equipment Corporation - Systems Research Center, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, 1994. available at http ://www.research.digital.com/SRC/.

[3] G. Aceto, B. bloom, and F. W. Vaandrager. Turning sos rules into equations. Technical Report CS-R9218, Computer Sciences/Department of Software Technology, CWI - Centrum voor Wiskunde en Informatica, 1992. http ://ftp.cwi.nl/cwi/publications/reports/reports.html.

[4] A. Aho, R. Sethi, and J. Ullman. *Compilers*. Addison-Wesley, 1986.

[5] M. Aksit, R. Mostert, and B. Haverkort. Compiler generation based on grammar inheritance. Technical Report 90-07, University of Twente, Department of Computer Science, P.O Box 217 7500 AE Enschede, the Netherlands, February 1990.

[6] J.-M. Andreoli, S. Freeman, and R. Pareschi. The coordination language facility : Coordination of distributed objects. *Theory and Practice of Object Systems*, 2(2), 1996.

[7] D. S. Arnon, I. Attali, and P. Franchi-Zannettacci. A document manipulation system based on natural semantics. *Mathematical and Computer Modelling Journal*, 1995.

[8] U. Assmann. Graph rewrite systems for program optimization. Technical Report RT-2955, Institut National de Recherche en Informatique et en Automatique, Août 1996. submitted to TOPLAS - available at http ://i44www.info.uni-karlsruhe.de/ assmann/optimix.html.

[9] U. Assmann. Optimix language report. Technical Report RT-0195, Institut National de Recherche en Informatique et en Automatique, Août 1996. available at http ://i44www.info.uni-karlsruhe.de/ assmann/optimix.html.

[10] I. Attali, D. Caromel, and S. O. Ehmety. an operational semantics for the eiffel// language. Technical Report RR-2732, Institut National de Recherche en Informatique et en Automatique, Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France), November 1995. http ://www.inria.fr/Unites/Sophia-fra.html (CROAP project).

[11] I. Attali and D. Parigot. Integrating natural semantics and attribute grammars : the minotaur system. Technical Report RR-2339, Institut National de Recherche en Informatique et en Automatique, Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France), November 1994. http ://www.inria.fr/Unites/Sophia-fra.html (CROAP/CHLOE project).

[12] M. Azuma, T. Tabata, Y. Oki, and S. Kamiya. Spd : A humanized documentation technology. *IEEE Computer*, 11(9) :945–953, September 1985.

[13] D. Ghazanfarpour B. Peroche, J. Argence and D. Michelucci. *La synthèse d'images*. ditions Hermès, 51, rue Rennequin, 75017, Paris, Janvier 1990.

[14] L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill. Distributed application configuration. *Proceedings of the 16th International Conference of Distributed Computing Systems, IEEE Computer Society, Hong Kong*, May 1996.

[15] J. Bertin. *La Graphique et le traitement graphique de l'information*. Flammarion, Juin 1977.

[16] J. Bertin. Représentation graphique. *Encyclopaedia Universalis, Corpus (8)*, pages 852–862, 1985.

[17] S. Bonin. *Initiation la graphique*. pi S.A. diteur, 1983.

[18] P. Borovansky, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 28(5), March 1997.

[19] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, and V. Pascual. Centaur : the system. In ACM, editor, *Proceedings of Sigsoft'88 third annual Symposium on Software Development Environments*, Boston, USA, October 1988.

[20] M. H. Brown and R. Sedgewick. A system for algorithm animation. In ACM Computer Graphics, editor, *Proceedings of SIGGRAPH'84, Minneapolis*, volume 18, USA, July 1984. ACM.

[21] K. B. Bruce. Progress in programming languages. *ACM Computing Surveys*, 28(1) :245–247, March 1996.

[22] L. Cardelli. Two dimensional syntax for functional languages. *Proceedings of Integrated Interactive Computing Systems - Elsevier Science Publishers*, pages 107–119, 1983.

[23] L. Cardelli. *Handbook of Computer Science and Engineering*, chapter Type Systems. CRC Press, 1997.

[24] L. Cardelli and P. Wegner. On understanding data types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17 :471–522, December 1985.

[25] A. Carle and L. Pollock. On the optimality of change propagation for incremental evaluation of hierarchical attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1) :16–29, January 1996.

[26] P. Carlson, M. Burnett, and J. Cadiz. A seamless integration of algorithm animation into a visual programming language. In ACM, editor, *Proceedings of AVI'96 : International Workshop on Advanced Visual Interfaces*, Gubbio, Italy, May 27-29 1996. ACM.

[27] N. Carriero and D. Gelernter. *How to Write Parallel Programs : A First Course*. MIT Press, 1990.

[28] Y. Caseau and F. Laburthe. Introduction to the claire programming language. Technical report, LIENS, 1996. http ://www.ens.fr/ laburthe/claire.html.

[29] P. Ciancarini. Personnal home page. http ://www.cs.unibo.it/ cianca.

[30] W. Citrin, R. Hall, and B. Zorn. Addressing the scalability problem in visual programming. Technical report, University of Colorado, Dpt of computer Science, April 1995.

[31] D. Clément. A distributed architecture for programming environments. In ACM, editor, *Proceedings of Sigsoft'90, the Fourth Symposium on Software Development Environments*, Irvine, USA, December 1990.

[32] P. Codognet and D. Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(1), 1996.

[33] J. Cohen. Logic programming and constraint logic programming. *ACM Computing Surveys*, 28(1), March 1996.

[34] A. Colmeraurer. Introduction to prolog-iii. *Communication of the ACM*, 33(7), July 1990.

[35] G. Costagliola, A. De Lucia, and S. Orefice. Toward efficient parsing of diagrammatic languages. In *Proceedings of Advanced Visual Interfaces (AVI'94)- Bari*, pages 162–171. ACM Press, June 1994. http ://www.unisa.it/gencos.dir/.

[36] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. Efficient parsing of data-flow graphs. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering (SEKE'95)- Rockville*, pages 226–233, June, 22-24 1995. http ://www.unisa.it/gencos.dir/.

[37] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A framework of syntactic models for the implementation of visual languages. In *Proceedings of the IEEE symposium on Visual Languages (VL'97)- Capri*, pages 58–65. IEEE, September 1997. http ://www.unisa.it/gencos.dir/vlcc.htm.

[38] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. Automatic generation of visual programming environments. *IEEE Computer*, 28(3) :56–66, March 1995.

[39] B. Courcelle. *Handbook of Theoretical computer Science*, chapter Graph Rewriting : An Algebraic and Logic Approach. Elsevier Science Publishers B.V., j. van leeuwen edition, 1990.

[40] J. Coutaz. *Interfaces Homme-Ordinateur, Conception et Réalisation*. DUNOD Informatique, 1990.

[41] P. T. Cox and T. Pietrzykowski. Using a pictorial representation to combine dataflow and object-orientation in a language independent programing mechanism. *IEEE Proceedings of Integrated International Computer Science conference*, pages 695–704, 1988.

[42] E. Csuhaj-varjú, J. Dassow, J. Kelemen, and G. Pàun. *Grammar Systems : A grammatical Approach to Distribution and Cooperation*, chapter 3. Cooperating Distributed Grammar Systems - the Basic Model. Gordon and Breach Science Publishers, 1994.

[43] P. Curtis. Constraint quantification in polymorphic type analysis. Technical Report CSL-90-1, Palo alto Reserch Center, 3333 Coyote Hill Rd. Palo Alto, California 94304, February 1990.

[44] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical computer Science*, chapter Rewrite Systems, pages 243–320. Elsevier Science Publishers B.V., j. van leeuwen edition, 1990.

[45] T. Despeyroux. Typol a formalism to implement natural semantics. Technical Report RR-2732, Institut National de Recherche en Informatique et en Automatique, Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France), November 1988. http ://www.inria.fr/Unites/Sophia-fra.html.

[46] B. Dion, L. Angeli, and A. Bravo Lastra. Paragraph : An interactive environment for parallelizing fortran programs. Technical Report RR-1920, Institut National de Recherche en Informatique et en Automatique, Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France), April 1993. http ://www.inria.fr/Unites/Sophia-fra.html.

[47] R. A. Duisberg. Animation temporal constraints : an overview of the animus system. In *Human Computer Interaction*, volume 3, 1988.

[48] H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and their Applications to Computer Sciences*, volume 532, pages 24–37, Heildeberg, 1990. Springer.

[49] F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello. Relation grammars : a grammatical model for a high-level specification of visual languages. In *Proceedings of AVI'96 : International Workshop on Theory of Visual Languages*, Gubbio, Italy, May 27-29 1996. ACM.

[50] D. Gelernter and L. Zuck. On what linda is : Formal description of linda as a reactive system. In LNCS, editor, *Proceedings of the Second International Conference on Coordination, Languages and Models (Coordination'97), Berlin, Germany*, volume 1282, pages 187–204, September 1997.

[51] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[52] B. Goldberg. Functional programming languages. *ACM Computing Surveys*, 28(1) :245–247, March 1996.

[53] J. M. Gooday and A. G. Cohn. Visual language syntax and semantics : A spatial logic approach. In *Proceedings of AVI'96 : International Workshop on Theory of Visual Languages*, Gubbio, Italy, May 27-29 1996. ACM.

[54] M. Gottlib. La rubrique à brac.

[55] Object Management Group. *CORBA : Architecture and Specification (Revision 2.0)*, July 1995.

[56] V. Haarslev. Formal semantics of visual languages using spatial reasoning. In *Proceedings of VL'95*, Darmstadt, Germany, 1995. IEEE symposium on Visual Languages.

[57] J. Hannan. Opérational semantics-directed compilers and machine architectures. *ACM Transactions on Programming Languages and Systems*, 16(4) :1215–1247, July 1994.

[58] H. Hartson and P. Gray. Temporal aspects of task in the user action notation. *Human-Computer Interaction*, 7 :1–45, 1992.

[59] J. Heering. Second order algebraic specification of static semantics. Technical Report CS-R9254, Computer Sciences/Department of Software Technology, CWI - Centrum voor Wiskunde en Informatica, 1992. http ://ftp.cwi.nl/cwi/publications/reports/reports.html.

[60] J. Heering, P.R.H. Hendriks, P. Klint, and J. Reckers. The syntax definition formalism sdf - reference manual. *SIGPLAN Notices*, 24(11) :43–75, 1989.

[61] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value passing. *Information and Computation*, 107 :202–236, 1993.

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

[63] International Standard ISO8807. Lotos - a formal description language based upon the temporal ordering of observational behaviour, 1988.

[64] J. F. Th. Kamperman and H. R. Walters. Arm abstract rewriting machine. Technical Report CS-R9218, Computer Sciences/Department of Software Technology, CWI - Centrum voor Wiskunde en Informatica, 1993. http ://ftp.cwi.nl/cwi/publications/reports/reports.html.

[65] J. F. Th. Kamperman and H. R. Walters. Simulating trss by minimal trss : a simple, efficient, and correct compilation technique. Technical Report CS-R9605, Computer Sciences/Department of Software Technology, CWI - Centrum voor Wiskunde en Informatica, 1996. http ://ftp.cwi.nl/cwi/publications/reports/reports.html.

[66] H. Kirchner. Some extensions of rewriting. In H. Comon and J.-P. Jouannaud, editors, *Term Rewriting, French Spring School of Theoretical Computer Science*, volume 909 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 54–73, Font Romeux (France), 1995. Also available as Technical Report 94-R-175, CRIN, Nancy (France).

[67] M. Kopache and E. P. Glinert. c^2 : A mixed textual/graphical environment for c. *IEEE Proceedings Workshop on Visual Languages*, pages 231–238, 1988.

[68] J. Kramer and J. Maggee. Exposing the skeleton in the coordination closet. In LNCS, editor, *Second International Conference on Coordination Languages and Models*, volume 1282, pages 18–31, September 1997.

[69] J. Lamping and R. Rao. The hyperbolic browser : A focus + context technique for visualizing large hierarchies. *Journal of Visual Languages and Computing- Academic Press*, 7 :33–55, 1996.

[70] L. Lamport. How to write a proof. Technical Report TR-94, Digital Equipment Corporation - Systems Research Center, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, 1993. available at http ://www.research.digital.com/SRC/.

[71] J.-M. Larchevêque. Optimal incremental parsing. *ACM Transactions on Programming Languages and Systems*, 17(1) :1–15, January 1995.

[72] J.-L. Lauriere. *Intelligence Artificielle, résolution de problèmes par l'homme et la machine*. Eyrolle, 1987.

[73] P. Levy. *L'ideographie dynamique*. les Editions du Concept Moderne, 13 Avenue de Ste Clotilde, 1205 Geneve, SUISSE, 1991.

[74] J. Longchamp, C. Godart, and J.-C. Derniame. Les environnements intégrés de production de logiciel. *Techniques et sciences informatiques*, 11(1) :31–95, 1992.

[75] F. Ludolph, Y. K. Chow, D. Ingalls, S. Wallace, and K. Doyle. The fabrik programming environment. *IEEE Proceedings of Workshop on Visual Languages*, pages 222–230, 1988.

[76] D. W. Macintyre. *Design and Implementation with Vampire*, chapter 7, pages 129–159. Manning Publications Co., 1995.

[77] J. Magee, J. Kramer, and N. Dullay. Darwin/mp : An environnement for parallel and distributed programming. *Proceedings of 26th HICSS, Vol. II (Software Technology)*, January 1993.

[78] B. J. McKenzie, C. Yeatman, and L. de Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4) :672–689, July 1995.

[79] B. Mélèze. Metal, un langage de spécification pour le système mentor. Technical Report RR-142, Institut National de Recherche en Informatique et en Automatique, Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France), Juin 1982. http ://www.inria.fr/Unites/Sophia-fra.html.

[80] Sun microsystems. the java programming language. http ://java.sun.com.

[81] R. Milner. *Communication and Concurrency*. Prentice Hall international, 66 Wood Lane End, Hemel Hempstead Hertfordshire, HP2 4RG UK, 1989.

[82] R. Milner. *Operational and Algebraic Semantic of Concurrent Processes*, volume 2, chapter Chap. 19, Handbook of Theoretical computer Science. Elsevier Science Publishers B.V., 1990.

[83] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (part 1 & 2). LFCS Laboratory for Foundation of Computer Sciences, University of Edinburgh, June 1989.

[84] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[85] J. C. Mitchell. *Handbook of Theoretical computer Science*, chapter Type Systems for Programming Languages. Elsevier Science Publishers B.V., j. van leeuwen edition, 1990.

[86] R. De Nicola and R. Pugliese. A process algebra based on linda. In LNCS, editor, *Proceedings of the First International Conference on Coordination, Languages and Models (Coordination'96), Cesena, Italy*, volume 1061, pages 160–178, April 1996.

[87] O. Nierstrasz. Regular types for active objects. In ACM press, editor, *OOPSLA'93 Conference Proceedings*, Washington D.C, USA, October 1993. Association for Computing Machinery, ACM Sigplan Notices.

[88] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 1998. available at http ://www.ucy.ac.cy/ucy/cs/george5.html.

[89] T. J. Parr. *Obtaining Practical Variants for LL(k) and LR(k) for $k > 1$ by Splitting Atomic Tuples*. PhD thesis, Purdue University, August 1993. available at http ://www.ANTLR.org/papers/parr.phd.thesis.ps.gz.

[90] T. J. Parr and R. W. Quong. Antlr : A predicated-ll(k) parser generator. *Software-Practice and Experience*, 25(7) :789–810, July 1995. http ://www.ANLTR.org/papers/antlr.ps.

[91] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Sciences Dept, Århus university, Denmark, 1981.

[92] T. A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems*, 17(1) :461–486, January 1995.

[93] Protheo, 1997. "www.loria.fr/equipe/protheo.html".

[94] J. Reckers and A. Schürr. A parsing algorithm for context-sensitive graph grammars. Technical report, Department of Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands, 1995.

[95] J. Reckers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing - Academic Press*, 8 :27–55, 1996.

[96] S. P. Reiss. Graphical programm developpement with pecan programm developpement system. *SIGPLAN Notices*, April 1984.

[97] S. P. Reiss. Pecan : Program development system tha support multiple views. *IEEE Transaction on Software Engineering*, 11(3) :276–285, March 1985.

[98] S. P. Reiss. Software tools and environments. *ACM Computing Surveys*, 28(1) :281–284, March 1996.

[99] T. Reps and T. Teitelbaum. The synthetizer generator. *Procerdings of ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environment*, pages 42–48, April 1984.

[100] G. Robertson, S. Card, and J. Mackinlay. Information visualization using 3d interactive animation. *Communication of the ACM*, 36(4) :57–71, April 1993.

[101] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1) :23–41, January 1965.

[102] G. Van Rossum. the python programming language. http ://www.python.org.

[103] M. Sarkar and M. H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12) :73–84, 1994.

[104] D. A. Schmidt. Programming language semantics. *ACM Computing Surveys*, 28(1) :245–247, March 1996.

[105] A. Schürr. Progress : A vhl -language based on graph grammars. In Rozenberg Ehrig, Kreowski, editor, *Proccedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, March 1990. LNCS 532.

[106] A. Schürr. Progres, a visual language and environment for programming with graph rewriting systems. Technical Report TR-AIB 94-11, Aachener Informatik-Berichte, 1994. http ://www-i3.informatik.rwth-aachen.de/research/progres/index.html.

[107] A. Schürr. http ://www-i3.informatik.rwth-aachen.de/research/progres/application.html, April 1998. (valid URL at this date).

[108] M. Shaw. Procedure calls are the assembly language of systems interconnection : Connectors deserve first-class statues. *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[109] J. T. Stasko. The path transition paradigm : A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3) :213–236, 1990.

[110] G. L. Steele. *Common LISP : the Language*. Digital Press, 1984.

[111] R. Switzer. *Eiffel : An Introduction*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[112] A. Teisman. *Introduction aux sciences cognitives*, chapter L'attention, les traits et la perception des objets, pages 153–191. Gallimard, collection Folio, Février 1992.

[113] W. Teitelman and L. Matinser. The interlisp programming environment. *IEEE Computer*, April 1981.

[114] M. Tucci, G. Vitello, and G. Costagliolia. Parsing nonlinear languages. *IEEE Transactions on Software Engineering*, 20(9) :720–739, September 1994.

[115] S. Üsküdarh. Generating visual editors for formally specified languages. In *Proceedings of the 10th International Symposium on Visual Languages*, St. Louis, Missouri, USA, 1994. IEEE.

[116] J.-Y. Vion-Dury. *Etude et réalisation d'un metteur au point graphique pour application réparties à objets*. Mémoire d'ingénieur - Conservatoire National des Arts et Métiers, Decembre 1993.

[117] J.-Y. Vion-Dury. *Application des techniques de visualisation aux systèmes à objets*. Diplôme d'Etudes Approfondies - Institut National Polytechnique de Grenoble, Mai 1994.

[118] J.-Y. Vion-Dury and F. Pacull. A structured workspace for a visual configuration language. In *Proceedings of VL'97*, Capri, Italy, 1997. IEEE symposium on Visual Languages.

[119] J.-Y. Vion-Dury and M. Santana. Virtual images : Interactive visualization of distributed object-oriented systems. In ACM press, editor, *OOPSLA'94 Conference Proceedings*, volume 29, pages 65–84, Portland,Oregon, USA, October 1994.

[120] E. Visser. A family of syntax definition formalism. In *Proceedings of ASF+SDF95, A workshop on Generating Tools from Algebraic Specifications*, May 1992.

[121] E. Visser. A family of syntax definition formalism. Technical Report 9504, Programming Research Group, CWI, Amsterdam, 1995.

[122] K. Wittenburg. *Recent Advances in Parsing Technologies*, chapter Predictive Parsing for Unordered Relational Languages. Kluwer, 1993.

[123] K. Wittenburg and L. Weitzman. Relational grammars : Theory and practice in a visual language interface for process modeling. In *AVI'96 International Workshop on Theory of Visual Languages*, Gubbio, Italy, May 1996. http ://www.cs.monash.edu.au/ berndm/TVL96/tvl96-home.html.

[124] A. Zündorf and A. Schürr. Nondeterministic control structures for graph rewriting systems. In *Proccedings of the 17th International Workshop on Graph-Theoretic concepts in computer Science*, 1991.

Annexes

Demonstrations

Proposition 4.35 En appliquant la définition [te-min], il vient

$$\frac{\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad (\gamma \rhd e_1 == e_2 : t \Rightarrow \gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\} \preccurlyeq t)}{\gamma \rhd e_1 == e_2 \preccurlyeq : \textbf{bool in } \{\textbf{true}, \textbf{false}\}}$$

En developpant le numérateur gauche, il vient deux arbres correspondant aux deux cas possibles $\gamma \rhd e_1 == e_2 \sim \textbf{true}$ et $\gamma \rhd e_1 == e_2 \sim \textbf{false}$ (application de [te-enum]) :

$$\frac{\gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e_1 == e_2 : \textbf{bool} \quad \dfrac{\gamma \rhd e_1 == e_2 \rightsquigarrow \textbf{true} \quad \gamma \rhd \textbf{true} \rightsquigarrow \textbf{true}}{\gamma \rhd e_1 == e_2 \sim \textbf{true}}\text{[te-equiv]}}{\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\}}$$

$$\frac{\gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd e_1 == e_2 : \textbf{bool} \quad \dfrac{\gamma \rhd e_1 == e_2 \rightsquigarrow \textbf{false} \quad \gamma \rhd \textbf{false} \rightsquigarrow \textbf{false}}{\gamma \rhd e_1 == e_2 \sim \textbf{false}}\text{[te-equiv]}}{\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\}}$$

avec

$$\frac{\dfrac{}{\gamma \rhd \textbf{bool}} \quad \dfrac{}{\gamma \rhd \textbf{true} : \textbf{bool}} \quad \dfrac{}{\gamma \rhd \textbf{false} : \textbf{bool}} \quad \gamma \rhd \textbf{true} \rightsquigarrow \textbf{true} \quad \gamma \rhd \textbf{false} \rightsquigarrow \textbf{false}}{\gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\}}\text{[t-enum]}$$

et

$$\frac{\dfrac{\dfrac{\gamma \rhd e_1 \preccurlyeq : t}{\gamma \rhd e_1 : t}\text{[te-env4]} \quad \dfrac{}{\gamma \rhd t \preccurlyeq \top}\text{[st-top]}}{\gamma \rhd e_1 : \top}\text{[st-sub]} \quad \dfrac{\dfrac{\gamma \rhd e_2 \preccurlyeq : t' \textbf{ in } S}{\gamma \rhd e_2 : t' \textbf{ in } S}\text{[te-env4]} \quad \dfrac{}{\gamma \rhd t' \textbf{ in } S \preccurlyeq \top}\text{[st-top]}}{\gamma \rhd e_2 : \top}\text{[st-sub]}}{\gamma \rhd e_1 == e_2 : \textbf{bool}}\text{[te-eq]}$$

qui constituent le developpement complet du numérateur gauche de notre conclusion. Soit t' tel que $\gamma \rhd e_1 == e_2 : t'$, nous devons montrer que $\gamma \rhd t' \preccurlyeq \textbf{bool in } \{\textbf{true}, \textbf{false}\}$. Il reste à examiner tous les cas ou le système d'inférence permet d'assigner un type à $e_1 == e_2$, sous les conditions exigées par les hypothèses.

1. cas $\gamma \rhd e_1 == e_2 : \textbf{bool}$

   $$\frac{\gamma \rhd e_1 : t \quad \gamma \rhd e_2 : t}{\gamma \rhd e_1 == e_2 : \textbf{bool}}\ \text{[te-eq]}$$

   (le developpement est déja réalisé dans l'arbre d'hypothèse). Il est facile de montrer

   $$\gamma \rhd e_1 == e_2 : \textbf{bool} \quad \Rightarrow$$

   $$\frac{\gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \dfrac{}{\gamma \rhd \textbf{true} : \textbf{bool}} \quad \dfrac{}{\gamma \rhd \textbf{false} : \textbf{bool}}}{\gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\} \preccurlyeq \textbf{bool}}\text{[st-enum]}$$

2. cas $\gamma \rhd e_1 == e_2 : \top$. Ce jugement est obtenu par

   $$\frac{\gamma \rhd e_1 == e_2 : \textbf{bool} \quad \gamma \rhd \textbf{bool} \preccurlyeq \top}{\gamma \rhd e_1 == e_2 : \top}\text{[st-sub]}$$

   l'implication est directe par [st-top].

3. cas $\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\}$. Ce typage est obtenu par

   $$\frac{\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}\} \quad \gamma \rhd \textbf{bool in } \{\textbf{true}, \textbf{false}\} \preccurlyeq \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\}}{\gamma \rhd e_1 == e_2 : \textbf{bool in } \{\textbf{true}, \textbf{false}, \textbf{none}\}}\text{[st-sub]}$$

   le resultat recherché est dans le numérateur.

Aucune autre règle ne peut être utilisée pour typer $e_1 == e_2$ (avec les hypothèses retenues).

Proposition 4.44 Par induction sur la structure des termes bien typés, toutes les transitions associées sont developpées. Nous produisons uniquement le cas $\gamma \rhd e : t$ **in** S.

L'hypothèse se développe de (manière unique) en

$$\frac{\gamma \rhd e : t \quad \dfrac{\gamma \rhd e_1 : t \cdots \gamma \rhd e_n : t}{\gamma \ \rhd \ t \textbf{ in } \{e_1, \cdots, e_n\}}[\text{t-enum}] \quad \exists i \in [1,n] \dfrac{\gamma \rhd e \rightsquigarrow v \quad \gamma \rhd e_i \rightsquigarrow v}{\gamma \rhd e \sim e_i}[\text{t-equiv}]}{\gamma \rhd e : t \textbf{ in } \{e_1, \cdots, e_n\}}[\text{te-enum}]$$

D'autre part, l'hypothèse d'induction nous permet d'écrire

$$\gamma \rhd e : t \quad \Rightarrow \quad \forall \mathcal{S} | \Vdash \mathcal{S} : \gamma, \ \ [\mathcal{S} \vdash e] \rightarrow [\mathcal{S}' \vdash e'], \ \ \Vdash \mathcal{S}' : \gamma, \ \ \gamma \rhd e' : t$$

Il reste donc a montrer que

$$\gamma \rhd e' : t \textbf{ in } \{e_1, \cdots, e_n\}$$

et plus particulierement,

$$\exists j \in [1,n] \text{ tel que } \gamma \rhd e' \rightsquigarrow e_j$$

Pour cela, nous utilisons l'hypothèse de convergence pour montrer que $j = i$, car $\gamma \rhd e \sim e_i \Rightarrow \gamma \rhd e' \sim e_i$ par

$$\frac{\dfrac{\Vdash \mathcal{S} : \gamma \quad [\mathcal{S} \vdash e] \twoheadrightarrow\circ \ [\mathcal{S}' \vdash v] \quad ([\mathcal{S} \vdash e] \rightarrow [\mathcal{S}'' \vdash e'] \Rightarrow \gamma \rhd e' \rightsquigarrow v)}{\gamma \rhd e \rightsquigarrow v}[\text{t-conv}] \quad \gamma \rhd e_i \rightsquigarrow v}{\gamma \rhd e \sim e_i}$$

$\Rightarrow$

$$\frac{\gamma \rhd e' \rightsquigarrow v \quad \gamma \rhd e_i \rightsquigarrow v}{\gamma \rhd e' \sim e_i}$$

Proposition 4.70

shéma de la démonstration . La preuve repose sur une induction portant à la fois sur les termes et les types. Une mesure de la complexité d'un terme ou d'un type, notée $|e|_\gamma$ et $|t|_\gamma$ est proposée. Dans un premier temps, un lemme établit que toute évaluation d'un terme bien typé e ou d'un type bien formé t produisant une forme normale v ou u est telle que $|v|_\gamma \leq |e|_\gamma$ et $|u|_\gamma \leq |t|_\gamma$. Dans un second temps, nous produisons l'hypothèse d'induction $\mathcal{H}_n$, puis nous montrons qu'elle est vérifiée pour $n = 0$. Ensuite, nous montrons que pour tout $i \geq 0$, $\mathcal{H}_i \implies \mathcal{H}_{i+1}$.

Mesure de complexité

$$
\begin{array}{lcl}
|\gamma| & = & \sup(|x_i|_\gamma, \ldots, |\tau_j|_\gamma) \\
 & & \forall x_i, \tau_j \in \text{ dom}(\gamma) \\
 & & \\
|t|_\gamma & = & 0 \\
 & & \text{si } t \in \{\textbf{string}, \textbf{num}, \textbf{None}, \textbf{bool}, \bot, \top\} \\
 & & \\
|t \textbf{ in } \{e_1, \cdots, e_n\}|_\gamma & = & \sup(|t|_\gamma, |e_1|_\gamma, \cdots, |e_n|_\gamma) + 1 \\
|t_1 \to t_2|_\gamma & = & \sup(|t_1|_\gamma, |t_2|_\gamma) + 1 \\
|\tau|_\gamma & = & |\gamma(\tau)|_\gamma \\
 & & \\
|x|_\gamma & = & |\gamma(x)|_\gamma \\
|v|_\gamma & = & 0 \\
 & & \text{si } v \in \{\textbf{n}, \textbf{s}, \textbf{true}, \textbf{false}, \textbf{none}\} \\
|e_1 \star e_2|_\gamma & = & \sup(|e_1|_\gamma, |e_2|_\gamma) + 1 \\
 & & \star \in \{+, \leq, , ==, \textbf{and}\} \\
|\textbf{not } e|_\gamma & = & |e|_\gamma + 1 \\
|e_1(e_2)|_\gamma & = & \sup(|e_1|_\gamma, |e_2|_\gamma) + 1 \\
|\lambda x : t.e|_\gamma & = & \sup(|t|_\gamma, |e|_{\gamma,x:t}) + 1 \\
|\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3|_\gamma & = & \sup(|e_1|_\gamma, |e_2|_\gamma, |e_3|_\gamma) + 1
\end{array}
$$

Lemme sur la réduction de termes et types confluents

Lemme A.1 *Convergence*

$$
\forall \gamma, \left\{ \begin{array}{lll}
\gamma \rhd e \rightsquigarrow v & \Longrightarrow & |v|_\gamma \leq |e|_\gamma \\
\gamma \ \rhd \ t \textit{ et pour tout } \mathcal{S} \textit{ tel que } \Vdash \mathcal{S} : \gamma \textit{ et } [\mathcal{S} \vdash t] \to\!\!\circ \ [\mathcal{S} \vdash u] & \Longrightarrow & |u|_\gamma \leq |t|_\gamma
\end{array} \right.
$$

Preuve : Om considère inductivement tous les cas. □

Hypothèse d'induction Nous posons l'hypothèse d'induction pour tout environnement bien formé γ et $n \geq 0$,

$$
\mathcal{H}_n \stackrel{def}{=} \left\{ \begin{array}{ll}
(a) & |\gamma| \leq n \\
(b) & \forall e, t, t_1, t_2, \\
 & \text{si } |e|_\gamma \leq n, |t|_\gamma \leq n, |t_1|_\gamma \leq n, |t_2|_\gamma \leq n \\
 & \text{alors} \left\{ \begin{array}{lll}
\mathcal{T}ype(\gamma, t) \equiv \textbf{true} & \Longrightarrow & \gamma \ \rhd \ t \\
\mathcal{F}nd(\gamma, e) \equiv t & \Longrightarrow & \gamma \rhd e \preccurlyeq : t \\
\mathcal{S}ub(\gamma, t_1, t_2) \equiv \textbf{true} & \Longrightarrow & \gamma \rhd t_1 \preccurlyeq t_2 \\
\mathcal{C}hk(\gamma, e, t) \equiv \textbf{true} & \Longrightarrow & \gamma \rhd e : t \\
\mathcal{A}ddType(\gamma, t_1, t_2) \equiv t & \Longrightarrow & \gamma \rhd t \doteq t_1 \oplus t_2 \\
\mathcal{P}rodType(\gamma, t_1, t_2) \equiv t & \Longrightarrow & \gamma \rhd t \doteq t_1 \otimes t_2
\end{array} \right.
\end{array} \right.
$$

Demonstration de $\mathcal{H}_o$ non saisie

Induction $\mathcal{H}_i \implies \mathcal{H}_{i+1}$ **pour** $i \geq 0$ non saisie

1. $t_1, t_2 \in \{\textbf{num}, \textbf{bool}, \textbf{None}, \textbf{string}, \bot, \top\}$.

Théorème 4.71 *non saisi*