HAL open science

Fonctions et généricité dans un langage de programmation parallèle

Jean-Michel Hufflen

▶ To cite this version:

Jean-Michel Hufflen. Fonctions et généricité dans un langage de programmation parallèle. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1989. Français. NNT : . tel-00335698

HAL Id: tel-00335698
https://theses.hal.science/tel-00335698

Submitted on 30 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée par

JEAN-MICHEL HUFFLEN

pour obtenir le titre de

DOCTEUR de l'**INSTITUT**
NATIONAL POLYTECHNIQUE de **GRENOBLE**

(arrêté ministériel du 5 juillet 1984)

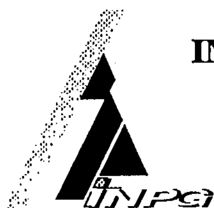
Spécialité : Informatique

Fonctions et généricité dans un langage de programmation parallèle

Thèse soutenue le 5 juillet 1989 devant la commission d'examen composée de :

Jean-Pierre VERJUS	Président
Jean-Pierre BANÂTRE **Claude KIRCHNER**	Rapporteurs
Didier BERT **Philippe JORRAND**	Examinateurs

Thèse préparée sous la direction de Ph. JORRAND,
au sein du LIFIA (**L**aboratoire d'**I**nformatique **F**ondamentale et d'**I**ntelligence **A**rtificielle)



INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

46 avenue Felix Viallet

38031 GRENOBLE cedex

Tél. : 76.57.45.00

Année universitaire 1989

Président de l'Institut :

Monsieur Georges LESPINARD

Professeurs des Universités

BARIBAUD Michel	ENSERG	JAUSSAUD Pierre	ENSIEG
BARRAUD Alain	ENSIEG	JOST Rémy	ENSPG
BAUDELET Bernard	ENSPG	JOUBERT Jean-Claude	ENSPG
BEAUFILS Jean-Pierre	INPG	JOURDAIN Geneviève	ENSIEG
BLIMAN Samuel	ENSERG	LACOUME Jean-Louis	ENSIEG
BOIS Philippe	ENSHMG	LADET Pierre	ENSIEG
BONNETAIN Lucien	ENSEEG	LESIEUR Marcel	ENSHMG
BONNET Guy	ENSPG	LESPINARD Georges	ENSHMG
BRISSONNEAU Pierre	ENSIEG	LONGEQUEUE Jean-Pierre	ENSPG
BRUNET Yves	IUFA	LORET Benjamin	ENSHMG
CAILLERIE Denis	ENSHMG	LOUCHET François	ENSEEG
CAVAIGNAC Jean-François	ENSPG	LUCAZEAU Guy	ENSEEG
CHARTIER Germain	ENSPG	MASSE Philippe	ENSIEG
CHENEVIER Pierre	ENSERG	MASSELOT Christian	ENSIEG
CHERADAME Hervé	UFR PGP	MAZARE Guy	ENSIMAG
CHERUY Arlette	ENSIEG	MOHR Roger	ENSIMAG
CHOVET Alain	ENSERG	MOREAU René	ENSHMG
COHEN Joseph	ENSERG	MORET Roger	ENSIEG
COLINET Catherine	ENSEEG	MOSSIERE Jacques	ENSIMAG
CORNUT Bruno	ENSIEG	OBLED Charles	ENSHMG
COULOMB Jean-Louis	ENSIEG	OZIL Patrick	ENSEEG
COUMES André	ENSERG	PA ULEAU Yves	ENSEEG
CROWLEY James	ENSIMAG	PERRET Robert	ENSIEG
DARVE Félix	ENSHMG	PIAU Jean-Michel	ENSHMG
DELLA-DORA Jean	ENSIMAG	PIC Etienne	ENSERG
DEPEY Maurice	ENSERG	PLATEAU Brigitte	ENSIMAG
DEPORTES Jacques	ENSPG	POUPOT Christian	ENSERG
DEROO Daniel	ENSEEG	RAMEAU Jean-Jacques	ENSEEG
DESRE Pierre	ENSEEG	REINISCH Raymond	ENSPG
DOLMAZON Jean-Marc	ENSERG	RENAUD Maurice	UFR PGP
DURAND Francis	ENSEEG	ROBERT André	UFR PGP
DURAND Jean-Louis	ENSPG	ROBERT François	ENSIMAG
FAUTRELLE Yves	ENSHMG	SABONNADIERE Jean-Claude	ENSIEG
FOGGIA Albert	ENSIEG	SAUCIER Gabrièle	ENSIMAG
FONLUPTJean	ENSIMAG	SCHLENKER Claire	ENSPG
FOULARD Claude	ENSIEG	SCHLENKER Michel	ENSPG
GANDINI Alessandro	UFR PGP	SERMET Pierre	ENSERG
GAUBERT Claude	ENSPG	SILVY Jacques	UFR PGP
GENTIL Pierre	ENSERG	SIRIEYS Pierre	ENSHMG
GENTIL Sylviane	ENSIEG	SOHM Jean-Claude	ENSEEG
GREVEN Hélène	IUFA	SOLER Jean-Louis	ENSIMAG
GUEGUEN Claude	ENSIEG	SOUQUET Jean-Louis	ENSEEG
GUERIN Bernard	ENSERG	TROMPETTE Philippe	ENSHMG
GUYOT Pierre	ENSEEG	VINCENT Henri	ENSPG
IVANES Marcel	ENSIEG	ZADWORNY François	ENSERG

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
COURNIL M.
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LACHENAL D.
LADET Pierre
LATOMBE Claudine
LE HUY H.
LE GORREC Bernard
MADAR Roland
MEUNIER G.
MULLER Jean
NGUYEN TRONG Bernadette
NIEZ J.J.
PASTUREL Alain
PLA Fernand
ROGNON J.P.
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri
YAVARI A.R.

Chercheurs du C.N.R.S

DIRECTEURS DE RECHERCHE CLASSE 0

LANDEAU	Ioan
NAYROLLES	Bernard

Directeurs de recherche 1ère Classe

ANSARA Ibrahim
CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
KRAKOWIAK Sacha
LEPROVOST Christian
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ARMAND Michel
AUDIER Marc
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
CHATILLON Chritiant
CLERMONT Jean-Robert
COURTOIS Bernard
DAVID René
DION Jean-Michel
DRIOLE Jean
DURAND Robert
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GARNIER Marcel
GUELIN Pierre
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MEUNIER Jacques
PEUZIN Jean-Claude
PIAU Monique
RENOUARD Dominique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
VENNEREAU Pierre
WACK Bernard
YONNET Jean-Paul

Personnalités agrées à titre permanent à diriger des travaux de recherche
(décision du conseil scientifique)

E.N.S.E.E.G
HAMMOU Abdelkader
MARTIN-GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.M.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIEB Maurice
VINCENDON Marc

Laboratoires extérieurs :

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Situation particulière

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J..Claude	Détachement	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement	30/09/1989
ENSPG	BLOCH	Daniel	Recteur à c/	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. NEMOZ Alain

Année Universitaire 1988 - 1989

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerde	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean-Pierre	Mécanique,
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pures

JOSELEAU Jean Paul	Biochimie
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI J.Pierre	STAPS
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean Pierre	Mécanique
COLOMBEAU Jean François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre Yves	Informatique
DAVID Jean	Géographie

DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodymamique CNRS - CRTBT
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean René	Physique
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
CHEHIKIAN Alain	E.E.A. I.U.T.1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2 ème classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHENAVAS Jean	Physique IUT 1

CHILO Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
FOSTER Panayotis	Chimie IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
HAMAR Roger	Chimie IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
LEVIEL Jean Louis	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
TURGEMAN Sylvain	Génie civil
VINCENDON Marc	Chimie IUT 1
ZIGONE Michel	Physique IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci

HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.

MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

La satisfaction d'avoir achevé cette thèse va de pair avec le plaisir de remercier tous ceux qui, au cours de ces années, ont assuré ma formation de chercheur, m'ont encouragé, critiqué, soutenu, conseillé...

En premier lieu, Philippe Jorrand, responsable du projet FP2, qui m'a accueilli dans son équipe, et qui, malgré un emploi du temps chargé de Directeur du Laboratoire, a suivi l'évolution de mon travail. Je le remercie de tous ses conseils et resterai sensible à la confiance qu'il m'a accordée.

Merci à Jean-Pierre Verjus, qui m'a fait l'honneur de présider le jury.

Je tiens à exprimer ma gratitude à Jean-Pierre Banâtre et Claude Kirchner, rapporteurs de cette thèse, pour leur lecture approfondie de la première version. La forme présente du document leur doit beaucoup.

Merci à Didier Bert, responsable du projet LPG, pour sa participation au jury. Elle est la suite de nombreuses discussions au cours desquelles nous avons comparé les langages LPG et FP2.

J'ai également plaisir à remercier les membres des équipes FP2 et LPG, collègues et néanmoins excellents camarades : Rachid Echahed, Philippe Schnoebelen, Hubert Comon, Xavier Pandolfi, Zoubir et Mounira Belmesk, Pascal Drabik, Jean-Claude Reynaud, Sadik Sebbar Alaoui, Paul Jacquet, Maria Blanca Ibáñez, Ramón Brena, Françoise Nayroles, Philippe Chatelin, Marie-Laure Potet, sans oublier les "anciens" : Guilherme Bittencourt, Denis Lugiez, Saddek Bensalem, Sylvie Rogé, Mary Cisneros, Juan Manuel Pereira, Maria Antonia Mozota.

J'associe à cette liste les chercheurs extérieurs qui se sont intéressés à mon travail, en particulier Daniel Le Métayer, Susanne Graf, Hubert Garavel, Olivier Raoult, pour leurs commentaires sur des extraits de la thèse. Mention spéciale pour Meryem Marzouki, qui a relu l'intégralité de la première version.

Durant ces années de préparation, j'ai bénéficié d'abord d'une bourse C^3, ensuite de contrats ESPRIT. Mes derniers remerciements, mais non les moindres, vont aux organismes qui ont assuré ma "survie matérielle".

Grenoble, juillet 1989.

Résumé

FP2 (**F**unctional **P**arallel **P**rogramming) est un langage qui concilie programmation fonctionnelle et programmation parallèle à travers le formalisme des spécifications algébriques et des systèmes de réécriture. Dans le cadre du projet FP2, cette thèse a pour principal objectif de présenter la partie fonctionnelle, incluant la généricité et le traitement des exceptions.

La généricité (paramétrisation d'une spécification) est traitée dans la première partie : nous rappelons les principes, étudions la sémantique, formalisons la compilation des opérateurs génériques en restant dans un cadre fonctionnel, et analysons les raccourcis de notation offerts aux utilisateurs.

La deuxième partie est consacrée aux exceptions. Elles sont d'abord étudiées d'un point de vue opérationnel, puis nous en donnons une définition précise qui permet de ramener une présentation avec exceptions à une présentation avec sortes ordonnées. Cette définition assure l'existence d'une algèbre initiale et permet en outre de traiter les exceptions avec paramètres génériques.

En troisième partie, sont présentées des méthodes de transformation de définitions fonctionnelles récursives en processus parallèles communicants. La généricité est utilisée pour formuler les hypothèses sur les définitions fonctionnelles, et nous montrons de plus comment simuler une pile de récursivité de profondeur arbitraire par des réseaux de processus dont la topologie est fixée statiquement.

Mots-clés : spécification algébrique, langage fonctionnel et parallèle, généricité, sémantique, compilation, exceptions, transformation de programmes.

Abstract

FP2 (**F**unctional **P**arallel **P**rogramming) is a language combining both functional and parallel programming through the algebraic specification approach and term rewriting systems. As part of the project FP2, the principal object of this thesis is the presentation of the functional part, including genericity and the treatment of exceptions.

Part I presents genericity (i.e. parameterization of a specification). We recall the principles, study the semantics, formalize the compilation of generic operators in a functional framework, and analyze the notational facilities provided to the users.

In part II, we investigate the exception mechanism. At first, we present them operationally. Then, we show that it is possible to define a presentation with exceptions as an order-sorted presentation. Thus, the existence of an initial algebra is ensured. Moreover, this method permits to treat exceptions with generic parameters.

In part III, methods for transforming recursive functional definitions into parallel programs are presented. The hypotheses on the functional definitions are specified by means of generic operators. Besides, we show how to simulate an arbitrarily deep recursive stack by means of networks with a statically fixed topology.

Keywords: algebraic specification, functional and parallel language, genericity, semantics, compilation, exceptions, program transformation.

La véritable science enseigne, par dessus-tout, à douter et à être ignorant.

MIGUEL de UNAMUNO (1864-1936)
Le sentiment tragique de la vie

Fonctions et généricité dans un langage de programmation parallèle

Jean-Michel HUFFLEN

Introduction

Un problème crucial de l'activité de programmmation est la correction : "le programme que nous venons d'écrire répond-il bien aux spécifications du problème posé ?" La réalisation d'applications de plus en plus ambitieuses, d'une complexité parfois difficilement maîtrisable, a clairement montré la nécessité d'une discipline pour atteindre, ou tout du moins approcher cet objectif primordial.

Il s'est vite avéré que les programmes impératifs étaient difficiles, voire impossibles à prouver. L'émergence de certaines méthodes de programmation (programmation structurée, programmation modulaire et compilation séparée), et de langages qui les favorisaient (PASCAL, MODULA-2, ADA), a permis de concevoir des programmes mieux construits, plus clairs et plus faciles à maintenir. Néanmoins, ces programmes continuaient à représenter des transitions d'état d'une machine de Von Neumann et il était extrêmement difficile de les considérer comme des objets, de concevoir à leur sujet des raisonnements élégants, au même titre que sur d'autres objets mathématiques.

S'est donc fait sentir le besoin d'un nouveau style de programmation, le besoin de nouveaux langages dont la sémantique serait davantage fondée sur la notion de fonction, plutôt que sur l'évolution de l'état d'une machine. Par opposition à la programmation *impérative* reposant entièrement sur l'opération d'affectation, on parle ainsi de programmation *fonctionnelle* ou *applicative*. Le précurseur de ce type de langage fut LISP pur [133], inspiré du λ-calcul d'Alonzo Church [39, 6], et introduit par John McCarthy. D'autres langages ont suivi. Citons FP (Functional Programming) [4], de John Backus, totalement affranchi de la notion de variable et introduisant les formes fonctionnelles : les arguments d'une forme fonctionnelle peuvent être des fonctions ou des objets, le résultat est une fonction. L'aspect extérieur du langage (définitions de fonctions par combinaisons de fonctions et de formes fonctionnelles) est souvent peu lisible, mais la formulation des fonctions se prête parfaitement aux transformations de programmes [177]. Signalons également le langage KRC (Kent Recursive Calculator) [173], qui fournit des constructeurs d'objets infinis (par exemple, "[1..]" représente la liste infinie des entiers naturels non nuls), et dont les techniques d'évaluation paresseuse des arguments des fonctions, apparues en [94], permettent des calculs sur ces objets infinis (par exemple, prendre le premier élément ou le reste d'une liste infinie). KRC est, de plus, un langage du second ordre : les fonctions peuvent admettre des fonctions comme arguments ou retourner une fonction comme résultat.

Cependant, un reproche important que l'on peut adresser à ces trois langages est qu'ils ne sont pas typés, du moins pas au sens d'un contrôle statique. Ainsi, ce n'est qu'à l'exécution que l'on vérifie qu'une addition arithmétique est bel et bien appliquée à deux arguments entiers. Si nous revenons aux langages impératifs traditionnels, ces vérifications de types sont effectuées statiquement. Par contre, la variété des types disponibles demeure assez pauvre, et leur utilisation sévèrement limitée. On ne peut paramétrer ces types : il est par exemple nécessaire, lorsqu'on écrit un parcours d'arbre, de préciser le type des éléments de l'arbre. Plus généralement, la définition de nouveaux types, quand elle est permise, est étroitement liée à leur représentation en machine : même remarque qu'en ce qui concerne

les preuves, à savoir que ces langages restent trop proches de l'architecture de Von Neumann. C'est pour corriger ces défauts tout en conservant les avantages des contrôles de type qu'ont été introduits les *types abstraits*. Un type abstrait est défini uniquement par son interface, par les fonctions que l'on souhaite appliquer aux éléments de ce type.

L'introduction de types abstraits dans le λ-calcul a ouvert la voie à toute une classe de langages dont le premier fut ML [89]. Ce langage permet de définir des types abstraits, éventuellement *polymorphes*, c'est-à-dire paramétrés par d'autres types. Ainsi il est possible, par exemple, d'écrire dans ce langage une fonction *reverse* qui inverse l'ordre d'apparition des éléments d'une liste, quel que soit le type de ces éléments. À cette famille de langages appartient HOPE [35], fortement modulaire. Le principe de son implantation est une compilation des programmes HOPE dans un code pour une machine abstraite à pile. MIRANDA [174] reprend l'approche du λ-calcul typé en y intégrant les principaux acquis de KRC. À signaler que la forme extérieure de langages comme ML ou MIRANDA est légère, car un mécanisme d'*inférence de types* [139] décharge l'utilisateur de l'indication du type d'un objet ou du profil d'une fonction. De nombreux ouvrages ont déjà été consacrés à l'implantation des ces langages, les plus récents se basant sur l'utilisation de combinateurs [45] et introduisant les machines de réduction de graphes [148]. D'une façon plus générale, le lecteur intéressé trouvera en [100] une présentation de la théorie des types, ainsi que des directions qui en découlent.

Tous les langages cités précédemment utilisent la notion de *réduction*, en tant que procédé de calcul. Une autre approche a cherché à privilégier l'aspect algébrique, la démarche de *spécification*. L'accent est alors mis davantage sur la description des problèmes que sur l'aspect algorithmique d'une solution. (Traitant plus généralement de méthodologie de la programmation, [132] présente l'étape de spécification comme une *abstraction* de toutes les implantations de cette spécification.) Là encore, a été introduite la possibilité de *paramétrer* une spécification par une autre spécification : on parle alors de spécifications *génériques*. Les principaux travaux sur les spécifications algébriques sont [87, 86] (travaux du groupe ADJ), ainsi que [33, 91, 93, 22, 59]. Un certain nombre de langages se sont inspirés de cette approche : CLEAR [34], LARCH [92], LPG (Langage de Programmation Générique) [17, 20, 19], PLUSS (Proposition pour un Langage Utilisant des Spécifications Structurées), langage de spécification de l'environnement ASSPEGIQUE (Assistance à la Spécification Algébrique) [24, 23], ainsi que les réalisations successives d'OBJ (OBJ1 [82], puis OBJ2 [63, 169] et OBJ3 [80, 74, 88]).

Un autre objectif de l'activité de programmation est l'efficacité. Même si les performances des calculateurs séquentiels continuent à s'améliorer, on compte actuellement davantage sur la programmation parallèle, l'exécution simultanée de plusieurs tâches sur des processeurs différents. Par ailleurs, il est important de remarquer que le développement des systèmes distribués fournit un intérêt supplémentaire à l'étude du parallélisme. Néanmoins, pour aussi prometteur que soit ce concept, il reste beaucoup à accomplir avant de le comprendre et de le maîtriser, et dans nombre de sujets : formalismes, langages, méthodes de programmation et outils d'aide à la mise au point, preuves de systèmes parallèles, transformations de programmes...

Si nous examinons les langages actuels qui permettent d'exprimer le parallélisme, plusieurs classifications existent : l'une d'entre elles consiste à distinguer les langages synchrones, fondés sur la présence d'une horloge (ESTEREL [14], LUSTRE [38, 149], SIGNAL [68]), des langages asynchrones (CCS — Calculus of Communicating Systems [140], LC^3 [130]). De même, on peut classer d'après le mécanisme de communication : par rendez-vous (CSP — Communicating Sequential Processes [96]), par mémorisation des messages dans une file d'attente (LC^3). Une troisième classification possible est de séparer les langages fondés sur une programmation impérative (CSP, ESTEREL, LC^3) de ceux fondés sur une programmation applicative (LUSTRE, SIGNAL). (Le lecteur intéressé par le détail de

ces classifications de langages parallèles pourra trouver en [147] un tour d'horizon sur ces langages et les méthodes de programmation qui leur sont sous-jacentes.)

Le projet à l'intérieur duquel prend place cette thèse est le développement du langage FP2 (**F**unctional **P**arallel **P**rogramming). À l'intérieur de toutes les classifications précédentes, FP2 est un langage applicatif, il appartient à la famille des langages parallèles sans horloge, et son principe de communication est fondé sur le rendez-vous à plusieurs ("à N"). C'est un langage de spécifications algébriques paramétrées, son propos est de concilier programmation fonctionnelle et programmation parallèle à travers le formalisme des spécifications algébriques et des systèmes de réécriture. On peut donc spécifier des types et des fonctions génériques en FP2. On peut également y spécifier des processus, et des réseaux de processus parallèles communicants, tous ces objets pouvant être paramétrés d'une manière analogue aux objets de la partie fonctionnelle, au moyen de la généricité.

Ce langage est le support d'études d'analyse de processus (terminaison [146], comparaison des comportements des processus [154], analyse temporelle [163], transformation de programmes [106]). Comme exemples de programmation "grandeur nature" en FP2, nous citerons un simulateur par événements [157], l'unification parallèle de deux termes [104], une simulation du niveau liaison du protocole X25 [8]. Enfin, dans le cadre du projet ESPRIT dans lequel se trouve inclus FP2, il est prévu de l'utiliser pour programmer une machine à inférence parallèle en logique du premier ordre [107], en relation avec les travaux sur la méthode de connexion [21]. FP2 est également un projet soutenu par le programme C^3 (**C**oopération, **C**oncurrence et **C**ommunication) du CNRS [175], consacré à la compréhension et à la maîtrise du concept de parallélisme.

Le prototype actuel regroupe l'implantation de la partie fonctionnelle et une simulation séquentielle des processus. Il fait suite à deux maquettes : l'une (μFP2 [161]) axée principalement sur la simulation des processus, l'autre ("Grand-Guignol" [105]) consacrée uniquement à la partie fonctionnelle. Dans l'avenir, il est prévu que viennent s'ajouter au prototype une partie graphique permettant aux utilisateurs de rentrer un réseau de processus sous forme d'un dessin, des outils de mise au point, et les implantations des travaux sur l'analyse et la comparaison des processus. Deux implantations parallèles du langage sont également prévues. L'une sera effectuée sur machine Stollmann, au moyen d'un réseau comportant un processeur maître et quelques autres processeurs. La compilation des processus se déroulera de la même manière que pour la simulation séquentielle. Ensuite, durant une exécution, le maître dirigera le calcul en distribuant aux autres processeurs des évaluations de termes [155]. L'autre implantation prévue utilisera les *transputers* et procèdera par transformation d'un processus FP2 en un processus simulable par un *transputer* [144], et transformation d'un réseau quelconque de processus en un réseau compatible avec l'architecture parallèle dont on dispose ([136] aborde le cas particulier du placement d'un réseau de processus FP2 sur une structure de grille).

Parmi toutes ces tâches, notre contribution concerne essentiellement la partie fonctionnelle et générique du langage, dont nous avons assuré l'étude théorique et l'implantation. Cette partie fonctionnelle est pour une large part inspirée de LPG — également développé au LIFIA —, quoiqu'elle s'en distingue par les principes de mise en œuvre. Les autres différences concernent le traitement des opérateurs partiels, et les notations, plus légères en FP2, plus orientées vers la programmation.

Nous présentons succintement FP2 dans un chapitre préliminaire, après une introduction aux concepts qui sont à la base du langage : la spécification algébrique de types abstraits et les systèmes de réécriture (tous les calculs, en FP2, s'effectuent par réécriture). Sont également rappelées dans ce

chapitre les bases théoriques du langage OBJ3, dont, au cours de la thèse, nous comparons les choix à ceux de FP2.

La généricité est approfondie ensuite, dans la première partie. Alors que les précédentes présentations de FP2 [110, 111] évoquaient la généricité de manière assez informelle, le premier chapitre regroupe l'étude opérationnelle et sémantique complète de la généricité, d'abord d'un point de vue théorique général, puis par l'application à FP2 (présentée de manière différente de [152]). Intuitivement, une spécification générique est caractérisée par la donnée d'une spécification *paramètre* et d'une spécification *cible*. L'*instanciation*, c'est-à-dire le *remplacement* d'une spécification *formelle* par une spécification *effective*, est effectuée au moyen de *morphismes* entre spécifications, qui en préservent la structure. Un avantage important de FP2 est la *génération* de morphismes entre spécifications : ceci permet de reporter de proche en proche une propriété (par exemple, d'une propriété d'ordre total sur les entiers, on peut déduire une propriété d'ordre total sur les séquences d'entiers, une propriété d'ordre total sur les séquences dont les éléments sont eux-mêmes des séquences d'entiers, ...). Nous proposons une sémantique de cette possibilité, déjà présente informellement en LPG.

Ensuite, le chapitre 2 est consacré à la mise en œuvre. La démarche suivie a été de *compiler* les opérateurs de FP2 en un langage purement fonctionnel, et non dans un code pour une machine virtuelle, comme en LPG. Ce choix permet de formaliser le procédé de compilation sans sortir du cadre fonctionnel (du point de vue de la réalisation, le langage cible est un sous-ensemble de Common Lisp). Étant donné que la paramétrisation d'une présentation fait intervenir non seulement des sortes formelles, mais aussi des opérateurs formels, le principal apport de ce chapitre est le traitement de ces opérateurs formels, ainsi que le traitement des opérateurs génériques paramétrés par ces opérateurs formels. Après un rappel des principes généraux qui ont guidé cette réalisation, nous donnons les règles de compilation des termes génériques, et la correction par rapport à la sémantique opérationnelle vue au premier chapitre.

Dans le chapitre 3, nous présentons les facilités offertes à l'utilisateur. L'instanciation des opérateurs génériques conduisant à des termes dont l'écriture est souvent complexe, nous fournissons des raccourcis de notation, qui permettent de décrire partiellement une instanciation, le compilateur se chargeant alors de compléter l'information, s'il peut le faire de façon univoque. Ces raccourcis sont présents en LPG, où ils sont décrits et utilisés informellement ; nous les avons généralisés en FP2 et nous donnons les règles précises d'analyse.

La deuxième partie (chapitre 4) est consacrée aux exceptions, qui ont été intégrées à FP2 pour permettre la spécification de fonctions partielles. Nous décrivons les mécanismes qui leur sont attachés : *déclenchement*, *propagation*, et *récupération*. Nous montrons ensuite que l'introduction d'exceptions non récupérées ne modifie pas la propriété de terminaison ou de non-terminaison d'un système de réécriture (ce théorème figure comme conjecture dans [150], où il a été démontré sous des hypothèses plus fortes). Nous poursuivons le chapitre par l'étude sémantique : après un rappel des concepts proposés pour le traitement des exceptions, nous proposons une définition qui permet de ramener une présentation avec exceptions à une présentation avec sortes ordonnées, telle que le comportement décrit opérationnellement soit correctement simulé par la présentation avec sortes ordonnées obtenue. Ce résultat dote les exceptions d'une sémantique qui nous semble satisfaisante, et permet en outre de traiter les exceptions avec paramètres génériques, ce qui n'existait pas jusqu'à présent. De plus, cette méthode fournit des niveaux emboîtés pour les études de propriétés : selon les cas, on pourra se placer dans le cadre des présentations, dans le cadre des présentations paramétrées, dans le cadre des présentations avec sortes ordonnées paramétrées. Le chapitre se termine par une proposition

de traitement local des exceptions pour les processus, et par une description de l'implantation des exceptions.

Nous avons également contribué à rapprocher les deux parties de FP2 en développant des méthodes de transformation de programmes fonctionnels à double appel récursif en processus parallèles communicants. C'est le point que nous développons dans la troisième partie (chapitre 5). Les outils de la partie générique sont utilisés pour exprimer nos hypothèses sur les définitions fonctionnelles. Outre des exemples de réseaux simulant le calcul de fonctions récursives, nous montrons comment simuler une pile de récursivité de profondeur arbitraire par des réseaux dont la topologie est fixée statiquement. Nous donnons une preuve informelle de notre transformation de base, et présentons une méthode de regroupement qui permet d'éviter les évaluations redondantes dans des fonctions telles que la fonction de Fibonacci.

Après une synthèse de nos résultats et des principes qui ont guidé notre démarche, nous indiquons en conclusion quelques poursuites possibles des travaux présentés dans cette thèse.

Nous rappelons dans l'annexe A les définitions de théorie des catégories que nous utilisons dans l'ouvrage, ces définitions étant illustrées par quelques exemples issus de la théorie des ensembles. L'annexe B regroupe des exemples détaillés d'utilisations de la partie fonctionnelle de FP2. Dans l'annexe C, nous donnons et démontrons un résultat de réécriture utilisé dans le chapitre 4. Quant à l'annexe D, elle précise comment s'effectue le dialogue avec l'utilisateur, au niveau des entrées-sorties, durant l'exécution d'un processus FP2.

Afin de rendre la consultation et la lecture plus aisées, la liste des exemples en FP2 mentionnés tout au long de la thèse est donnée en p. 241. Nous avons également joint un index comprenant les principales dénominations et notations utilisées.

Chapitre 0

Notions de base — Présentation succinte de FP2

Ce chapitre préliminaire poursuit un triple but : introduire les notions fondamentales et la terminologie de base de la spécification algébrique, présenter succintement FP2, et également introduire une généralisation qui n'est pas directement intégrée en FP2, mais que nous utiliserons au chapitre 4 — de plus, cette généralisation constitue la base théorique du langage OBJ3 [80, 125, 126, 74, 88, 117], dont, à plusieurs reprises, nous comparerons les choix aux nôtres, notamment en ce qui concerne l'importation de définitions et la mise en œuvre de la généricité. Ainsi sont regroupées ci-après la plupart des définitions classiques liées à la théorie de la spécification algébrique, que nous utiliserons tout au long de l'ouvrage. La présentation de FP2 nous permettra d'effectuer un premier tour d'horizon sur les caractéristiques du langage et les possibilités qu'il offre à ses utilisateurs. Aussi bien pour la partie "programmation fonctionnelle" que pour la partie "programmation parallèle", nous commençons par décrire le cas particulier non paramétré (nous verrons plus tard que nous pouvons le comprendre comme une paramétrisation *triviale*), puis nous introduisons informellement la généricité. L'étude de cette dernière sera ensuite reprise et approfondie dans la première partie de la thèse. En fait, nous présentons sommairement la généricité dans ce chapitre afin de montrer dès à présent ce qu'elle peut apporter aux spécifications de processus communicants. Ceux-ci interviendront à nouveau aux chapitres 4 et 5. Historiquement, FP2 est issu des travaux présentés dans [110, 40, 1, 146, 115]. [111, 114, 160] marquent l'étape suivante dans la définition du langage. [165], plus récent, propose une nouvelle sémantique de la partie parallélisme. Mentionnons également [113, 128], qui sont des présentations des projets dans lesquels est inclus FP2. Le survol qui suit reprend les principes et les notations de l'implantation réalisée en 1987–1988, et intègre les principaux acquis de [165].

0.1 Programmation fonctionnelle

0.1.1 Les bases théoriques de la spécification algébrique

Les fondements théoriques de la partie fonctionnelle de FP2 sont les spécifications algébriques de types abstraits [33, 87, 86, 91, 93, 22, 59] et les systèmes de réécriture [102, 98, 118, 51, 52]. Ainsi que nous l'avons mentionné dans l'introduction, l'idée de base des *types abstraits* est d'ignorer leur représentation, de ne les considérer qu'au travers des opérations qui leur sont appliquées. C'est cette idée qui est reprise ici, dans le cadre de la spécification algébrique. Par conséquent, pour chaque type qu'il définit, l'utilisateur d'un langage de spécification algébrique ne connaît que le nom par lequel il

le désigne, les opérateurs présents dans son interface, et les équations qui caractérisent ces opérateurs. Nous rappelons les définitions, pour la plupart classiques.

0.1.1.1 Signatures — Algèbres hétérogènes

Définition 0.1 *Une* **signature algébrique** *— ou, plus simplement,* **signature** *— est un couple* (S, Ω) *où* S *est un ensemble non vide de* **sortes** *(*noms *de types), et* Ω *une famille indicée par* $S^* \times S$ *d'ensembles d'***opérateurs**.

Nous convenons de noter un mot de $S^* \times S$ par $s_1 \times \cdots \times s_n \to s$, et de noter un élément ω de $\Omega_{s_1 \times \cdots \times s_n \to s}$ par $\omega : s_1 \times \cdots \times s_n \to s$. $s_1 \times \cdots \times s_n \to s$ est alors le **profil** de ω, $s_1 \times \cdots \times s_n$ son **domaine**, s son **codomaine** et n son **arité**. Le domaine d'un opérateur peut être vide (noté "$-$", $\omega : - \to s$), auquel cas l'opérateur est dit **constant**.

Exemple 0.2 *Signature comportant les sortes Bool et Nat.*

$$S = \{Bool, Nat\}$$

$$\begin{aligned} \Omega_{-\to Bool} &= \{true, false\} & \Omega_{-\to Nat} &= \{zero\} \\ \Omega_{Bool\to Bool} &= \{not\} & \Omega_{Nat\to Nat} &= \{succ\} \\ \Omega_{Bool\times Bool\to Bool} &= \{and, or, \Rightarrow, \Leftrightarrow\} & \Omega_{Nat\times Nat\to Bool} &= \{=\} \end{aligned}$$

Introduisons à présent les *modèles sémantiques* d'une signature, et les homomorphismes entre ces objets, qui en préservent la structure.

Définition 0.3 *Soit* $\Sigma = (S, \Omega)$ *une signature. Une* **algèbre hétérogène** A *sur* Σ *— ou, plus simplement,* Σ**-algèbre** *— est la donnée de deux familles* (A_S, A_Ω) *— une famille d'ensembles non vides et une famille de fonctions*[1] *totales — telles que :*

$$\begin{aligned} A_S &= \{A_s \mid s \in S\} \\ A_\Omega &= \{A_\omega : A_{s_1} \times \cdots \times A_{s_n} \to A_s \mid \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}, s_1, \ldots, s_n, s \in S\} \end{aligned}$$

Les éléments de A_S *sont les* **supports** *de* A.

Les deux applications suivantes :

$$\begin{aligned} \iota : S &\to A_S & \iota : \Omega &\to A_\Omega \\ s &\mapsto A_s & \omega &\mapsto A_\omega \end{aligned}$$

constituent l'**interprétation** de Σ dans A.

Définition 0.4 *Soit* $\Sigma = (S, \Omega)$ *une signature, soient* A *et* B *deux* Σ*-algèbres. Un* Σ**-homomorphisme** $h : A \to B$ *est la donnée d'une famille d'applications* $\{h_s : A_s \to B_s \mid s \in S\}$*, telles que :*

(i) $(\forall \omega \in \Omega_{-\to s}), h_s(A_\omega) = B_\omega$

(ii) $(\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}), (\forall x_i \in A_{s_i}, 1 \leqslant i \leqslant n), h_s(A_\omega(x_1, \ldots, x_n)) = B_\omega(h_{s_1}(x_1), \ldots, h_{s_n}(x_n))$

Proposition 0.5 *Soit* Σ *une signature. Les* Σ*-algèbres et les* Σ*-homomorphismes forment les objets et les flèches d'une catégorie*[2] *notée* $\mathbf{ALG}_\Sigma$.

[1]Nous utilisons le mot "opérat<u>eur</u>" pour désigner un symbole *syntaxique* et les mots "opérat<u>ion</u>" ou "fonct<u>ion</u>" pour les concepts *sémantiques*.

[2]Nous avons regroupé à l'annexe A toutes les définitions et tous les résultats utilisés de la théorie des catégories.

0.1.1.2 Termes

Par combinaison des divers opérateurs d'une signature, on construit des termes. Nous allons indiquer ci-après comment engendrer des termes *bien formés.*

Définition 0.6 *Soient $\Sigma = (S, \Omega)$ une signature et V ($V \cap \Omega = \emptyset$) une famille indicée par S d'ensembles disjoints de* **variables**[3]. *L'***algèbre des termes engendrée par** *V, notée $T_\Sigma(V)$, est la réunion des ensembles de* **termes** *$T_{\Sigma_s}(V)$, pour s parcourant S, construits comme suit (les notations "$\overline{\omega}$" et "$\overline{\omega(t_1, \ldots, t_n)}$" indiquent que ces expressions formées au moyen d'un opérateur ω et des termes $t_1, \ldots, t_n$ doivent être interprétées* syntaxiquement*) :*

(i) $V_s \subseteq T_{\Sigma_s}(V)$

(ii) $(\forall \omega \in \Omega_{_ \rightarrow s}), \overline{\omega} \in T_{\Sigma_s}(V)$

(iii) $(\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \rightarrow s}), (\forall t_i \in T_{\Sigma_{s_i}}(V), 1 \leqslant i \leqslant n), \overline{\omega(t_1, \ldots, t_n)} \in T_{\Sigma_s}(V)$

(iv) $(\forall s \in S)$, *tout élément de $T_{\Sigma_s}(V)$ s'obtient à partir des règles (i), (ii) et (iii).*

$T_\Sigma(\emptyset)$ est appelé l'**algèbre des termes** de Σ et est notée plus simplement T_Σ.

D'après la définition 0.6, il est immédiat d'introduire $sort(t)$, la **sorte** d'un terme t :

$$(\forall s \in S), (\forall t \in T_{\Sigma_s}(V)), sort(t) \stackrel{\text{déf}}{=} s$$

et $var(t)$, l'ensemble des variables d'un terme t :

$$\begin{array}{rcll} var(x) & = & \{x\} & (x \in V) \\ var(\overline{\omega}) & = & \emptyset & \\ var(\overline{\omega(t_1, \ldots, t_n)}) & = & \bigcup_{1 \leqslant i \leqslant n} var(t_i) & \end{array}$$

Si $var(t) = \emptyset$, le terme t est dit **fermé**.

Définition 0.7 ([168]) *Soient $\Sigma = (S, \Omega)$ une signature, et $s \in S$. s est dite* **habitée** *s'il existe au moins un terme fermé de sorte s. Σ est dite* **complètement habitée** *si toutes les sortes de S sont habitées.*

Dans toute la suite, nous ne considérerons que des signatures complètement habitées.

Nous allons signaler deux propriétés importantes, l'une, concernant $T_\Sigma(V)$, liée à la théorie des algèbres universelles [25, 90], et l'autre, concernant T_Σ, qui se rattache au formalisme des catégories.

Soient $\Sigma = (S, \Omega)$ une signature, A une Σ-algèbre, et V un ensemble de variables. On appelle **assignation** toute application $\nu : V \rightarrow A$ qui préserve les sortes, c'est-à-dire telle que :

$$(\forall s \in S), \nu(V_s) \subseteq A_s$$

Proposition 0.8 *$T_\Sigma(V)$ est une* **Σ-algèbre libre** *sur V, c'est-à-dire que pour toute Σ-algèbre A et toute assignation $\nu : V \rightarrow A$, il existe un Σ-homomorphisme unique, défini sur $T_\Sigma(V)$, qui étend ν.*

[3] Par analogie avec les opérateurs, nous notons $x : s$ une variable x de sorte s. Dans toute la suite, nous considérerons, par abus de langage, V comme un ensemble de variables. De plus, chaque fois que nous aurons besoin de travailler avec des variables, nous supposerons que pour tout $s \in S$, V_s est infini (autrement dit, nous aurons à disposition autant de variables que nous le souhaiterons).

Cette extension s'effectue d'après les règles suivantes :

$$\begin{array}{rcl} \nu(\overline{\omega}) & = & \iota(\omega) \\ \nu(\overline{\omega(t_1,\ldots,t_n)}) & = & \iota(\omega)(\nu(t_1),\ldots,\nu(t_n)) \end{array}$$

$\iota(\omega)$ désignant l'interprétation de l'opérateur ω dans la Σ-algèbre A. $\nu(t)$ est une **instance** du terme t.

Note Par abus de langage, nous avons désigné par A à la fois l'algèbre et la réunion de tous ses ensembles. Toujours par abus de langage, dans les exemples que nous rencontrerons, nous conserverons le même symbole pour noter à la fois l'application et son extension à $T_\Sigma(V)$.

Proposition 0.9 *T_Σ est une Σ-algèbre initiale dans la catégorie* $\mathbf{ALG}_\Sigma$.

Nous abordons maintenant quelques concepts liés aux termes, que nous utiliserons par la suite. Nous commençons par introduire un moyen formel de désigner les emplacements des divers sous-termes d'un terme : des mots d'entiers naturels appelés occurrences.

Définition 0.10 *Soit t un terme, les* **occurrences** *(ou* **positions***) de t forment l'ensemble suivant (λ représente le mot vide) :*

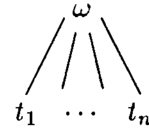
$$occ(t) = \begin{cases} \{\lambda\} & (t \in V) \vee (t = \overline{\omega}, \omega \in \Omega) \\ \{\lambda\} \cup (\bigcup_{1 \leqslant i \leqslant n} \{i.\tau \mid \tau \in occ(t_i)\}) & (t = \overline{\omega(t_1,\ldots,t_n)}) \end{cases}$$

Soit $\tau \in occ(t)$, le **sous-terme de** t **à l'occurrence** τ*, noté t/τ, est défini comme suit :*

$$\begin{array}{rcl} t/\lambda & = & t \\ \overline{\omega(t_1,\ldots,t_n)}/i.\tau & = & t_i/\tau \end{array}$$

On représente parfois un terme par un arbre étiqueté. Par analogie avec cette représentation, on appelle **racine** (*root*) d'un terme la variable ou le symbole d'opérateur en tête :

$$\begin{array}{rcll} root(t) & = & t & (t \in V) \vee (t = \overline{\omega}, \omega \in \Omega) \\ root(\overline{\omega(t_1,\ldots,t_n)}) & = & \omega & \end{array}$$



et **profondeur** (*depth*) d'un terme la longueur de sa plus longue occurrence :

$$depth(t) = \max\{\lg(\tau) \mid \tau \in occ(t)\}$$

Définition 0.11 *Soient t et t' deux termes. Soit $\tau \in occ(t)$ tel que $sort(t/\tau) = sort(t')$. Le terme issu de t en remplaçant son sous-terme à l'occurrence τ par t', noté $t[\tau \leftarrow t']$, est défini comme suit :*

$$\begin{array}{rcl} t[\lambda \leftarrow t'] & = & t' \\ \overline{\omega(t_1,\ldots,t_i,\ldots,t_n)}[i.\tau \leftarrow t'] & = & \overline{\omega(t_1,\ldots,t_i[\tau \leftarrow t'],\ldots,t_n)} \end{array}$$

Remarque $sort(t[\tau \leftarrow t']) = sort(t)$

Définition 0.12 *Un terme est dit* **linéaire** *si chacune de ses variables ne possède qu'une seule occurrence.*

Nous indiquons à présent comment remplacer une variable par un terme, et ceci à chacune de ses occurrences. Puis nous rappelons ce qu'est la résolution d'équations entre termes, appelée unification.

Définition 0.13 *Une* **substitution** σ *est un endomorphisme*[4] *de* $T_\Sigma(V)$*, égal à l'identité sur* V *excepté un nombre* fini *de variables qui constituent le* **domaine** *de* σ *:*

$$D(\sigma) = \{x \mid x \in V \land \sigma(x) \neq x\}$$

L'ensemble des substitutions de $T_\Sigma(V)$ *est noté* $\mathbf{Subst}(T_\Sigma(V))$.

Nous rappelons les règles qui permettent de définir σ sur tout l'ensemble $T_\Sigma(V)$, à partir de la donnée de sa définition sur son domaine et de la propriété d'algèbre libre de $T_\Sigma(V)$:

$$\begin{aligned} \sigma(\overline{\omega}) &= \overline{\omega} \\ \sigma(\overline{\omega(t_1,\ldots,t_n)}) &= \overline{\omega(\sigma(t_1),\ldots,\sigma(t_n))} \end{aligned}$$

Remarque σ étant un endomorphisme de $T_\Sigma(V)$, il préserve la sorte d'une variable :

$$(\forall x \in V), sort(\sigma(x)) = sort(x)$$

L'ensemble des **variables introduites** par σ est noté $I(\sigma)$ et défini comme suit :

$$I(\sigma) = \bigcup_{x \in D(\sigma)} var(\sigma(x))$$

On définit une relation d'ordre partiel sur les substitutions par :

$$\sigma_1 \leq \sigma_2 \iff (\exists \sigma' \in \mathbf{Subst}(T_\Sigma(V))), \sigma' \circ \sigma_1 = \sigma_2$$

Définition 0.14 *Deux termes* t_1 *et* t_2*, de* $T_\Sigma(V)$*, sont dits* **unifiables** *si et seulement s'il existe une substitution* σ *telle que* $\sigma(t_1) \equiv \sigma(t_2)$ *— "*$\equiv$*" désignant l'égalité* syntaxique *entre termes.*

Proposition 0.15 **([153])** *Si* t_1 *et* t_2 *sont unifiables, alors il existe une substitution* ϖ*, de domaine contenu dans* $var(t_1) \cup var(t_2)$*, telle que :*

$$(\forall \sigma \in \mathbf{Subst}(T_\Sigma(V)) \textit{ telle que } \sigma(t_1) \equiv \sigma(t_2)), \varpi \leq \sigma$$

ϖ *est unique à un renommage*[5] *bijectif des variables près, c'est un* **plus petit unificateur** *de* t_1 *et* t_2.

De plus, on peut toujours choisir ϖ idempotente ($\varpi \circ \varpi = \varpi$, ou $D(\varpi) \cap I(\varpi) = \emptyset$ — les deux conditions sont équivalentes) et n'introduisant pas de nouvelles variables ($I(\varpi) \subseteq var(t_1) \cup var(t_2)$).

Dans toute la suite, la proposition "t_1 et t_2 sont unifiables et un plus petit unificateur est ϖ" sera notée :

$$\boxed{t_1 \overset{\varpi}{\approx} t_2}$$

[4] Un **endomorphisme** d'une algèbre X est un homomorphisme de X vers X.

[5] Un **renommage** est une substitution telle que l'image de toute variable est une variable.

Les applications de l'unification sont nombreuses et vont de l'intelligence artificielle aux bases de données, en passant par la programmation logique [42, 41] et la démonstration de théorèmes. Signalons dès à présent son utilisation pour le calcul de l'inférence de types dans des langages tels que ML [89] ou MIRANDA [174].

Divers algorithmes calculant un plus petit unificateur de deux termes ont déjà été proposés : par J. Robinson [153], par G. Huet [97], par M. Paterson et M. Wegman [145], par A. Martelli et U. Montanari [135], par F. Fages [60], par J. Corbin et M. Bidoit [44]. Mentionnons également [129], qui traite en profondeur la notion de plus petit unificateur, et [124], qui est une étude systématique d'outils pour les algorithmes d'unification, lorsque des opérateurs vérifient certaines propriétés (associativité, commutativité, idempotence, ...).

Définition 0.16 *Soient t_1 et t_2 deux termes. On dit que t_1* **filtre** *t_2 si et seulement s'il existe une substitution σ telle que $\sigma(t_1) = t_2$.*

Par analogie avec notre notation pour l'unification, nous écrirons :

$$\boxed{t_1 \overset{\sigma}{\precsim} t_2}$$

0.1.1.3 Équations

Nous allons à présent nous intéresser aux relations entre les divers termes de $T_\Sigma(V)$, ces relations étant exprimées au moyen d'équations.

Définition 0.17 *Soit Σ une signature. Une* **Σ-équation** *est une paire de termes $(t_1 == t_2)$, t_1 et $t_2 \in T_\Sigma(V)$, telle que :*

$$sort(t_1) = sort(t_2)$$

Nous allons maintenant caractériser les Σ-algèbres dans lesquelles une équation représente une égalité. Ceci s'effectue à l'aide de la notion d'assignation vue au §0.1.1.2, et, plus précisément, de l'extension d'une assignation, grâce à la propriété d'algèbre libre de $T_\Sigma(V)$.

Définition 0.18 *Soient Σ une signature et A une Σ-algèbre. La Σ-équation $(t_1 == t_2)$ est* **valide** *dans A si et seulement si pour toute assignation $\nu : var(t_1) \cup var(t_2) \to A$, alors :*

$$\overline{\nu}(t_1) = \overline{\nu}(t_2)$$

On note : $A \models (t_1 == t_2)$.

Définition 0.19 *Une* **présentation algébrique** *— ou, plus simplement,* **présentation** *— est un triplet (S, Ω, E) où $\Sigma = (S, \Omega)$ est une signature et E un ensemble de Σ-équations.*

Définition 0.20 *Soient (S, Ω, E) une présentation et A une algèbre hétérogène sur $\Sigma = (S, \Omega)$. A* **satisfait** *E si toutes les Σ-équations de E sont valides dans A :*

$$(A \models E) \Longleftrightarrow (\forall e \in E), A \models e$$

(On dit aussi que A est un **modèle** *de E.)*

Définition 0.21 *Soient Σ une signature et E un ensemble de Σ-équations. La* **plus petite congruence sur T_Σ qui contient** *E, notée "$\equiv_E$", s'obtient par fermeture des propriétés de :*

- *réflexivité,*
- *symétrie,*
- *transitivité,*
- *précongruence :*

$$(\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}), (\forall u_i, v_i \in T_{\Sigma_{s_i}}, 1 \leqslant i \leqslant n), (u_i \equiv_E v_i) \Longrightarrow \overline{\omega(u_1, \ldots, u_n)} \equiv_E \overline{\omega(v_1, \ldots, v_n)}$$

- *stabilité par substitution :*

$$(\forall u, v \in T_\Sigma), (u == v) \in E \Longrightarrow (\forall \sigma \in \mathbf{Subst_g}(T_\Sigma(V))), \sigma(u) \equiv_E \sigma(v)$$

$\mathbf{Subst_g}(T_\Sigma(V))$ *désignant les substitutions* fermées *(*ground substitutions*), c'est-à-dire à valeurs dans T_Σ.*

Définition 0.22 *Soit $\mathcal{P}$ une présentation, composée d'une signature Σ et d'un ensemble E de Σ-équations, et soit V un ensemble de variables. L'***algèbre des termes engendrée par** *V* **et qui satisfait** *E est l'algèbre quotient de $T_\Sigma(V)$ par "$\equiv_E$" :*

$$T_{\Sigma/\equiv_E}(V) \stackrel{\text{déf}}{=} T_\Sigma(V)/\equiv_E$$

Nous la noterons également $T_\mathcal{P}(V)$.

$T_{\Sigma/\equiv_E}(\varnothing)$ est l'**algèbre des termes qui satisfait** E et est notée plus simplement $T_{\Sigma/\equiv_E}$ ou $T_\mathcal{P}$.

Proposition 0.23 *$T_{\Sigma/\equiv_E}(V)$ est une algèbre libre sur V.*

Proposition 0.24 *Soit $\mathcal{P}$ une présentation, composée d'une signature Σ et d'un ensemble E de Σ-équations. Les Σ-algèbres qui satisfont E et les Σ-homomorphismes forment les objets et les flèches d'une catégorie notée $\mathbf{ALG}_{\Sigma/\equiv_E}$ ou $\mathbf{ALG}_\mathcal{P}$. $T_{\Sigma/\equiv_E}$ est une algèbre initiale de $\mathbf{ALG}_{\Sigma/\equiv_E}$.*

Suivant la démarche de [86], nous définissons donc la sémantique d'une présentation (Σ, E) comme étant la classe[6] des algèbres initiales de $\mathbf{ALG}_{\Sigma/\equiv_E}$. Désormais, par abus de langage, nous désignerons, pour toute sorte s, le **type** s comme étant l'interprétation dans l'algèbre initiale (à un isomorphisme près) de s — nous réserverons le mot "sorte" pour le concept syntaxique.

0.1.1.4 Réécriture : les bases

Nous verrons que la plupart des équations manipulées par FP2 sont *orientées* de manière à ce que les calculs s'effectuent par réécriture. Intuitivement, cette technique est fondée sur le remplacement à l'intérieur d'un terme, et ce *modus operandi* est itéré jusqu'à l'obtention d'un terme irréductible. Nous le formalisons ci-après.

[6] Pour une définition précise de cette notion, cf. annexe A, note 80, p. 178.

Définition 0.25 *Une* **règle de réécriture** *est un couple de termes* $(t_1 \longrightarrow t_2)$, t_1 *et* $t_2 \in T_\Sigma(V)$, *tel que :*

$$\begin{array}{rcl} sort(t_1) & = & sort(t_2) \\ var(t_1) & \supseteq & var(t_2) \end{array}$$

Un **système de réécriture** *est un ensemble* fini *de règles de réécriture*[7].

Exemple 0.26 *Le système de réécriture suivant décrit la fonction factorielle.*

$$\begin{array}{rcl} 0! & \longrightarrow & 1 \\ succ(n)! & \longrightarrow & succ(n) * (n!) \end{array}$$

Définition 0.27 *Soit* $\mathcal{R}$ *un système de réécriture. Un terme* t_1 *se* **réécrit** *en le terme* t_2 *dans* $\mathcal{R}$ *si et seulement si :*

$$(\exists \tau \in occ(t_1)), (\exists (l \longrightarrow r) \in \mathcal{R}), l \overset{\sigma}{\lesssim} t_1/\tau \wedge t_2 = t_1[\tau \leftarrow \sigma(r)]$$

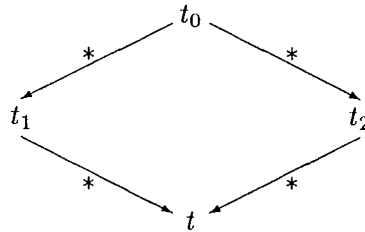
La proposition "t_1 se réécrit en t_2" se note $t_1 \longrightarrow_{\mathcal{R}} t_2$, ou plus simplement $t_1 \longrightarrow t_2$, lorsqu'il n'y a aucune ambiguïté sur le système de réécriture employé. "$\overset{*}{\longrightarrow}_{\mathcal{R}}$" est la fermeture réflexive et transitive de la relation "$\longrightarrow_{\mathcal{R}}$".

S'il est impossible de réécrire un terme, on dit qu'il est **irréductible** ou en **forme normale**.

Pour qu'un tel procédé soit cohérent, il est important qu'il soit indépendant des règles choisies parmi celles du système, en particulier indépendant de l'ordre d'application de ces règles. C'est ce que traduit la propriété de confluence.

Définition 0.28 *Un système de réécriture est* **confluent** *si et seulement si :*

$$(\forall t_0, t_1, t_2 \in T_\Sigma(V) \text{ tels que } t_0 \overset{*}{\longrightarrow}_{\mathcal{R}} t_1 \wedge t_0 \overset{*}{\longrightarrow}_{\mathcal{R}} t_2), (\exists t \in T_\Sigma(V)), t_1 \overset{*}{\longrightarrow}_{\mathcal{R}} t \wedge t_2 \overset{*}{\longrightarrow}_{\mathcal{R}} t$$



Proposition 0.29 *Si le système de réécriture* $\mathcal{R}$ *est confluent, alors la forme normale de tout terme, si elle existe, est unique.*

Pour un terme t, on la note $t{\downarrow}_{\mathcal{R}}$ ou plus simplement $t{\downarrow}$ lorsque $\mathcal{R}$ est univoque.

Définition 0.30 *Un système de réécriture est* **nœthérien** *s'il ne peut générer aucune chaîne infinie de réécriture :*

$$\nexists(t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow \cdots)$$

[7] Les auteurs de [52] admettent les systèmes de réécriture à nombre infini de règles.

On prouve qu'un système de réécriture $\mathcal{R}$ est nœthérien en exhibant un ordre "$\succsim$" sur $T_\Sigma(V)$, bien fondé[8], et qui vérifie $\xrightarrow{*}_{\mathcal{R}} \subseteq \succsim$.

Définition 0.31 *Un système de réécriture confluent et nœthérien est dit* **convergent**.

Nous signalons à présent une propriété importante pour l'utilisation de la réécriture à fins de preuves. Notons "$\xleftrightarrow{*}_{\mathcal{R}}$" la fermeture réflexive, symétrique et transitive de la relation "$\longrightarrow_{\mathcal{R}}$".

Définition 0.32 *Un système de réécriture $\mathcal{R}$ possède la* **propriété de Church-Rosser** *si :*

$$(\forall t_1, t_2 \in T_\Sigma(V) \textit{ tels que } t_1 \xleftrightarrow{*}_{\mathcal{R}} t_2), (\exists t \in T_\Sigma(V)), t_1 \xrightarrow{*}_{\mathcal{R}} t \xleftarrow{*}_{\mathcal{R}} t_2$$

L'intérêt de cette propriété : il est dans ce cas possible de démontrer un théorème de la forme $t_1 \xleftrightarrow{*}_{\mathcal{R}} t_2$ en utilisant uniquement des réécritures.

Proposition 0.33 *Un système de réécriture possède la propriété de Church-Rosser si et seulement s'il est confluent.*

0.1.1.5 Réécriture conditionnelle

Le mécanisme de réécriture, tel que nous venons de le décrire, se révèle insuffisant dans certaines applications. En effet, il peut être indispensable d'ajouter une condition pour qu'une réécriture soit licite, faute de quoi elle pourrait engendrer des termes incorrects (opérateurs partiels appliqués en dehors de leur domaine de définition), ou boucler indéfiniment. Deux exemples sont fournis par la division et la simplification des rationnels :

$$\frac{\frac{x}{y}}{\frac{z}{t}} \longrightarrow \frac{x * t}{y * z} \quad \text{if } (y \neq 0) \wedge (z \neq 0) \wedge (t \neq 0) \qquad \frac{x}{y} \longrightarrow \frac{x \text{ div } (x \text{ gcd } y)}{y \text{ div } (x \text{ gcd } y)} \quad \text{if } (y \neq 0) \wedge (x \text{ gcd } y \neq 1)$$

Dans le premier cas, les sous-termes de la conjonction protègent contre toute division par 0, dans le second cas, la condition "$x \text{ gcd } y \neq 1$" est indispensable afin de ne pas réécrire en lui-même un rationnel déjà sous forme irréductible. Après un bref rappel des notions de prédicat et d'atome, nous allons d'abord préciser la notion d'équation conditionnelle sous une forme très générale reprise de [142], donner sa sémantique, puis introduire le concept de réécriture conditionnelle[9].

Nous renvoyons à [166] le lecteur désireux d'approfondir les principes de la logique du premier ordre et du calcul des prédicats du premier ordre. Nous rappelons ci-après les notions qui nous seront utiles pour la réécriture conditionnelle.

Définition 0.34 *Soient $\Sigma = (S, \Omega)$ une signature, Π une famille indicée par S^* d'ensembles de symboles de* **prédicats**, *et V un ensemble de variables. L'ensemble des* **atomes** *sur $T_\Sigma(V)$, noté $At_\Sigma(V)$, est construit comme suit :*

(i) $(\forall \pi \in \Pi_-), \overline{\pi} \in At_\Sigma(V)$

---

[8]Un ordre est **bien fondé** s'il n'existe aucune chaîne infinie strictement décroissante : $\nexists(x_0 > x_1 > \ldots > \ldots)$.

[9]Dans [142], Peter Padawitz présente la réécriture conditionnelle comme un cas particulier d'une règle utilisée en démonstration automatique : la *paramodulation*. La même démarche se retrouve dans la thèse de Michaël Rusinowitch [156].

(ii) $(\forall \pi \in \Pi_{s_1 \times \cdots \times s_n}), (\forall t_i \in T_{\Sigma_{s_i}}(V), 1 \leqslant i \leqslant n), \overline{\pi(t_1, \ldots, t_n)} \in At_\Sigma(V)$

(iii) Tout élément de $At_\Sigma(V)$ s'obtient à partir des règles (i) et (ii).

Par analogie avec les opérateurs d'une signature, on parle du **domaine** et de l'**arité** d'un prédicat.

Pour toute Σ-algèbre A, une **interprétation** de Π dans A est la donnée d'une fonction ι telle que pour chaque $\pi \in \Pi_{s_1 \times \cdots \times s_n}$:

$$\iota(\pi) : A_{s_1} \times \cdots \times A_{s_n} \to \{true, false\}$$

Soit V un ensemble de variables, on étend aux atomes de $At_\Sigma(V)$ une assignation $\nu : V \to A$ de manière analogue aux termes, c'est-à-dire au moyen des règles :

$$\begin{aligned} \nu(\overline{\pi}) &= \iota(\pi) \\ \nu(\overline{\pi(t_1, \ldots, t_n)}) &= \iota(\pi)(\nu(t_1), \ldots, \nu(t_n)) \end{aligned}$$

Définition 0.35 *Soient Σ une signature, A une Σ-algèbre, et $P \in At_\Sigma(V)$. A* **satisfait** *P si pour toute assignation $\nu : V \to A$, alors :*

$$\nu(P) = true$$

On note : $A \models P$.

Définition 0.36 *Soient $\Sigma = (S, \Omega)$ une signature, Π une famille indicée par S^* de symboles de prédicats, et V un ensemble de variables. Une* **équation conditionnelle** *se note $(t_1 == t_2 \text{ if } P)$, avec $t_1, t_2 \in T_\Sigma(V)$, et $P \in At_\Sigma(V)$, tels que :*

$$sort(t_1) = sort(t_2)$$

P est la **prémisse** *(également appelée* condition *ou* contexte*) de l'équation conditionnelle.*

Définition 0.37 *Soient Σ une signature et A une Σ-algèbre. L'équation conditionnelle $(t_1 == t_2 \text{ if } P)$ est* **valide** *dans A si et seulement si pour toute assignation $\nu : var(t_1) \cup var(t_2) \cup var(P) \to A$, alors :*

$$(A \models \nu(P)) \Longrightarrow \nu(t_1) = \nu(t_2)$$

Remarque En remplaçant "$\nu(P)$" par "*true*" dans l'expression précédente, on retrouve bien évidemment la définition 0.18 de validité d'une équation inconditionnelle.

Pour rendre exécutables de telles équations, il convient non seulement d'introduire une orientation entre les membres des équations, mais également de restreindre les prémisses utilisées, afin de pouvoir préciser le moyen de les valider. Nous suivons l'approche de [150, 180, 27, 26] : l'utilisation du type *Bool* sous-jacent aux valeurs de la logique (cf. exemple 0.2), et ne considérons que des termes de sorte *Bool*. Il est alors nécessaire que le support de cette sorte soit isomorphe à $\{true, false\}$. Dans ce but, nous introduisons des définitions concernant le type *Bool*, après rappel de la construction de la plus petite congruence contenant un ensemble d'équations conditionnelles.

Proposition 0.38 ([150]) *Soient Σ une signature et E un ensemble d'équations conditionnelles. La* **plus petite congruence sur T_Σ qui contient** *E ("$\equiv_E$") peut être construite par approximations successives :*

$$\equiv_E = \bigcup_{i \geqslant 0} \equiv_{E,i}$$

où :

- *"$\equiv_{E,0}$" est la plus petite congruence sur T_Σ, engendrée par les équations de E qui sont de la forme $(t_1 == t_2)$,*
- *soit "$\sim_i$", pour $i \geqslant 0$, et $u, v \in T_\Sigma$, la relation définie par :*

$$u \sim_i v \Longleftrightarrow \left\{ \begin{array}{c} u \equiv_{E,i} v \\ \vee \\ (\exists (t_1 == t_2 \text{ if } c) \in E), (\exists \sigma \in \mathbf{Subst_g}(T_\Sigma(V))), \left\{ \begin{array}{ccc} \sigma(c) & \equiv_{E,i} & true \\ \sigma(t_1) & = & u \\ \sigma(t_2) & = & v \end{array} \right. \end{array} \right.$$

alors "$\equiv_{E,i+1}$" s'obtient à partir de "$\sim_i$" par fermeture des propriétés de réflexivité, symétrie, transitivité, précongruence et stabilité par substitution.

Remarque Cette construction est en fait une application de la théorie du point fixe [167].

Définition 0.39 *Une présentation (S, Ω, E), avec $S \ni Bool$ et E étant un ensemble d'équations conditionnelles, est dite :*

- **complète par rapport aux booléens** *si pour tout terme fermé t de sorte $Bool$, $t \equiv_E true$ ou $t \equiv_E false$,*
- **consistante par rapport aux booléens** *si $true \not\equiv_E false$.*

Dans la suite du §0.1.1.5, nous ne considérerons plus que des présentations vérifiant ces propriétés.

Théorème 0.40 ([150]) *Soient $\Sigma = (S, \Omega)$ une signature et E un ensemble d'équations conditionnelles. $T_{\Sigma/\equiv_E}$ est une algèbre initiale de $\mathbf{ALG}_{\Sigma/\equiv_E}$.*

Nous suivons le même plan que dans l'étude précédente des équations et systèmes de réécriture : pour définir à présent la réécriture conditionnelle, il nous reste à introduire une orientation des équations et une condition d'inclusion sur les variables.

Définition 0.41 *Une* **règle de réécriture conditionnelle** *est un triplet de termes de la forme $(l \longrightarrow r \text{ if } c)$, avec $l, r, c \in T_\Sigma(V)$, tels que :*

$$\begin{array}{rcl} sort(l) & = & sort(r) \\ sort(c) & = & Bool \\ var(l) & \supseteq & var(r) \cup var(c) \end{array}$$

Remarque Dans la définition 0.36, nous n'avons imposé aucune condition sur les variables d'une prémisse. Ainsi, dans l'équation conditionnelle :

$$f(x) == f'(x) \text{ if } g(x, y)$$

y a un sens *existentiel*. La condition d'inclusion des variables est également imposée à la prémisse dans le but d'éviter, lors de la réécriture, la recherche d'éventuelles valeurs, pour de telles variables, qui rendent la prémisse vraie.

Définition 0.42 *Soit $\mathcal{R}$ un système de réécriture conditionnelle. Un terme t_1 se réécrit en le terme t_2 dans $\mathcal{R}$ si et seulement si :*

$$(\exists \tau \in occ(t_1)), (\exists (l \longrightarrow r \text{ if } c) \in \mathcal{R}), l \overset{\sigma}{\lesssim} t_1/\tau \wedge \sigma(c) \overset{*}{\longrightarrow}_{\mathcal{R}} true \wedge t_2 = t_1[\tau \leftarrow \sigma(r)]$$

Note La définition précédente est récursive. Une règle telle que :

$$f(x) \longrightarrow a \text{ if } f(x) = a$$

est donc permise et boucle indéfiniment. On évite un tel comportement par l'introduction de niveaux hiérarchiques tels que la prémisse d'une règle de niveau i puisse être entièrement évaluée au niveau $i-1$. Cet aspect de la réécriture conditionnelle ne sera pas traité ici et nous renvoyons le lecteur intéressé à [150, 180, 27, 26]. À signaler que [120, 122] proposent une approche moins contraignante.

Les notions de système nœthérien, confluent, se généralisent à la réécriture conditionnelle. Signalons qu'une autre approche [120, 95, 121, 122] considère des règles de la forme :

$$l \longrightarrow r \text{ if } \bigwedge_{1 \leqslant i \leqslant n} (u_i == v_i)$$

avec :

$$\begin{array}{rrcl} & sort(l) & = & sort(r) \\ (\forall i : 1 \leqslant i \leqslant n), & sort(u_i) & = & sort(v_i) \\ & var(l) & \supseteq & var(r) \cup (\bigcup_{1 \leqslant i \leqslant n} (var(u_i) \cup var(v_i))) \end{array}$$

Un terme t_1 se réécrit en le terme t_2 suivant une telle règle si et seulement si :

$$(\exists \tau \in occ(t_1)), l \overset{\sigma}{\precsim} t_1/\tau \wedge (\bigwedge_{1 \leqslant i \leqslant n} (\exists w_i \text{ tel que } \sigma(u_i) \overset{*}{\longrightarrow}_{\mathcal{R}} w_i \overset{*}{\longleftarrow}_{\mathcal{R}} \sigma(v_i))) \wedge t_2 = t_1[\tau \leftarrow \sigma(r)]$$

Cette approche revient à se placer dans le cadre de la logique avec égalité. Même si nos conventions sont plus restrictives, nous avons préféré la première approche parce que c'est celle qui est utilisée en pratique, tout du moins pour le traitement des opérateurs partiels de FP2 (cf. chapitre 4), la condition de consistance par rapport aux booléens étant supposée satisfaite. De même, ce cadre nous suffit pour notre utilisation de la réécriture conditionnelle à l'annexe C.

0.1.2 Une généralisation : les algèbres avec sortes ordonnées

Les algèbres hétérogènes "classiques", c'est-à-dire *multi-sortes*, se révèlent parfois un formalisme trop rigide, compliquant le traitement de certaines applications. Ainsi, les seules relations entre les supports des sortes sont celles fournies par les fonctions de l'algèbre. Par conséquent, il n'existe aucun moyen direct de spécifier qu'un type est inclus dans un autre, si ce n'est par un opérateur représentant l'injection canonique. De même, si nous comprenons un opérateur f_2 comme représentant un prolongement de la fonction désignée par un opérateur f_1, il n'existe aucun moyen direct d'utiliser les équations définissant f_2 dans la spécification de f_1.

L'idée de base est d'introduire un ordre partiel sur les sortes. On parle alors d'algèbres avec sortes ordonnées (*order-sorted algebras*) par opposition aux algèbres hétérogènes multi-sortes (*many-sorted algebras*). On peut ainsi spécifier les types *Nat*, *Int* (entiers relatifs), *Rat* (rationnels), *GInt* (entiers de Gauss[10]) avec :

$$Nat < Int \begin{array}{l} < Rat \\ < GInt \end{array}$$

[10]Les entiers de Gauss forment l'ensemble :

$$\mathbb{Z}[i] = \{a + ib \mid a, b \in \mathbb{Z}\} \qquad (i^2 = -1)$$

ordre correspondant aux inclusions bien connues en Mathématiques. Si nous définissons :

$$\begin{array}{lcl} + & : & (Nat \times Nat) \rightarrow Nat \\ + & : & (Int \times Int) \rightarrow Int \\ + & : & (Rat \times Rat) \rightarrow Rat \\ + & : & (GInt \times GInt) \rightarrow GInt \end{array}$$

ces opérateurs seront eux aussi partiellement ordonnés par une relation de prolongement.

Les langages OBJ2 et OBJ3 permettent l'utilisation de sortes ordonnées[11]. Ils possèdent la même sémantique mathématique, mais les principes de mise en œuvre sont différents : OBJ2 [79, 63, 169, 64] utilise les résultats d'un théorème de *réduction* [79, 85] et transforme une spécification avec sortes ordonnées en une spécification multi-sorte (donc sans relation entre les différentes sortes) qui lui est équivalente, tandis qu'OBJ3 [80, 126, 88] exécute directement une réécriture avec sortes ordonnées.

Nous allons donner les définitions, reprises de [79, 63, 84, 126, 85], qui généralisent celles du §0.1.1 au cas des sortes ordonnées.

0.1.2.1 Signatures avec sortes ordonnées — Modèles sémantiques de Joseph Goguen et José Meseguer

Définition 0.43 *Une* **signature avec sortes ordonnées** *est un triplet* $\Sigma = (S, \leq, \Omega)$ *où* S *est un ensemble non vide de sortes,* Ω *une famille indicée par* $S^* \times S$ *d'ensembles d'opérateurs, et "*$\leq$*" un ordre partiel sur* S.

Dans toute la suite du §0.1.2, nous désignerons par "signature" une signature avec sortes ordonnées.

Définition 0.44 (Première définition d'une Σ-algèbre) *Soit* $\Sigma = (S, \leq, \Omega)$ *une signature. Une* **Σ-algèbre** A *est la donnée d'une famille d'ensembles et d'une famille de fonctions totales :*

$$\begin{array}{lcl} A_S & = & \{A_s \mid s \in S\} \\ A_\Omega & = & \{A_\omega : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s \mid \omega \in \Omega_{s_1 \times \cdots \times s_n \rightarrow s}, s_1, \ldots, s_n, s \in S\} \end{array}$$

qui satisfont les propriétés suivantes de compatibilité avec "$\leq$*" :*

(i) $(\forall s_1, s_2 \in S), s_1 \leq s_2 \Longrightarrow A_{s_1} \subseteq A_{s_2}$

(ii) $(\forall s'_1, \ldots, s'_n, s', s''_1, \ldots, s''_n, s'' \in S$ *tels que* $\bigwedge_{1 \leqslant i \leqslant n} (s'_i \leq s''_i) \wedge s' \leq s'')$*, alors :*

$$(\forall \omega \in \Omega_{s'_1 \times \cdots \times s'_n \rightarrow s'} \cap \Omega_{s''_1 \times \cdots \times s''_n \rightarrow s''}),$$
$$A_\omega : A_{s''_1} \times \cdots \times A_{s''_n} \rightarrow A_{s''} \text{ est un prolongement de } A_\omega : A_{s'_1} \times \cdots \times A_{s'_n} \rightarrow A_{s'}$$

Définition 0.45 (Première définition d'un Σ-homomorphisme) *Soit* Σ *une signature, et soient* A *et* B *deux* Σ*-algèbres. Un* **Σ-homomorphisme** $h : A \rightarrow B$ *est la donnée d'une famille d'applications* $\{h_s : A_s \rightarrow B_s \mid s \in S\}$*, telles que :*

(i) $(\forall \omega \in \Omega_{__\rightarrow s}), h_s(A_\omega) = B_\omega$

(ii) $(\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \rightarrow s}), (\forall x_i \in A_{s_i}, 1 \leqslant i \leqslant n), h_{s_1}(A_\omega(x_1, \ldots, x_n)) = B_\omega(h_{s_1}(x_1), \ldots, h_{s_n}(x_n))$

(iii) $(\forall s_1, s_2 \in S), s_1 \leq s_2 \Longrightarrow h_{s_1}$ *est la restriction à* A_{s_1} *de* h_{s_2}

Proposition 0.46 *Soit* Σ *une signature. Les* Σ*-algèbres et les* Σ*-homomorphismes forment les objets et les flèches d'une catégorie notée* $\mathbf{OSA}_\Sigma$.

[11] La sémantique de la première version d'OBJ (OBJ1 [82]) est fondée sur les algèbres d'erreur [71] (cf. §4.4.3).

0.1.2.2 Algèbre des termes

Définition 0.47 *Soient $\Sigma = (S, \leq, \Omega)$ une signature et V un ensemble de variables. L'***algèbre des termes engendrée par** *V, notée $\mathcal{T}_\Sigma(V)$, est la réunion des ensembles de termes $\mathcal{T}_{\Sigma_s}(V)$, pour s parcourant S, et construits comme suit :*

(i) $V_s \subseteq \mathcal{T}_{\Sigma_s}(V)$

(ii) $(\forall \omega \in \Omega_{__\to s}), \overline{\omega} \in \mathcal{T}_{\Sigma_s}(V)$

(iii) $(\forall \omega \in \Omega_{s_1\times\cdots\times s_n \to s}), (\forall t_i \in \mathcal{T}_{\Sigma_{s_i}}(V), 1 \leqslant i \leqslant n), \overline{\omega(t_1,\ldots,t_n)} \in \mathcal{T}_{\Sigma_s}(V)$

(iv) $(\forall s_1, s_2 \in S, s_1 \leq s_2), \mathcal{T}_{\Sigma_{s_1}}(V) \subseteq \mathcal{T}_{\Sigma_{s_2}}(V)$

(v) $(\forall s \in S)$, *tout élément de $\mathcal{T}_{\Sigma_s}(V)$ s'obtient à partir des règles (i), (ii), (iii) et (iv).*

$\mathcal{T}_\Sigma(\varnothing)$ est noté plus simplement $\mathcal{T}_\Sigma$.

Remarque Si l'on "oublie" la relation d'ordre et qu'on construit, pour chaque $s \in S$, $T_{\Sigma,}(V)$ d'après la définition 0.6 d'une algèbre de termes sur une signature multi-sorte, en général, on n'a ni $\mathcal{T}_{\Sigma,}(V) = \mathcal{T}_{\Sigma_s}(V)$, ni $\mathcal{T}_{\Sigma_s}(V) = \bigcup_{s' \leq s} T_{\Sigma_{s'}}(V)$.

Il résulte de la définition 0.47 que la sorte d'un terme, au sens où nous l'avons définie au §0.1.1.2, peut ne pas être unique. La notion intéressante à dégager ici est celle de *plus petite sorte* d'un terme, lorsqu'elle existe. C'est dans ce but qu'ont été introduites les signatures régulières.

Définition 0.48 *Une signature $\Sigma = (S, \leq, \Omega)$ est* **régulière** *si et seulement si :*

$$(\forall s_1, \ldots, s_n, s'_1, \ldots, s'_n, s' \in S \text{ tels que } \bigwedge_{1\leqslant i\leqslant n} (s_i \leq s'_i)), (\forall \omega \in \Omega_{s'_1\times\cdots\times s'_n \to s'}), \text{ alors :}$$

$$(\exists(\check{s}_1, \ldots, \check{s}_n, \check{s}), \text{ unique et minimal dans } S^{n+1}), \bigwedge_{1\leqslant i\leqslant n} (s_i \leq \check{s}_i) \land \omega \in \Omega_{\check{s}_1\times\cdots\times\check{s}_n \to \check{s}}$$

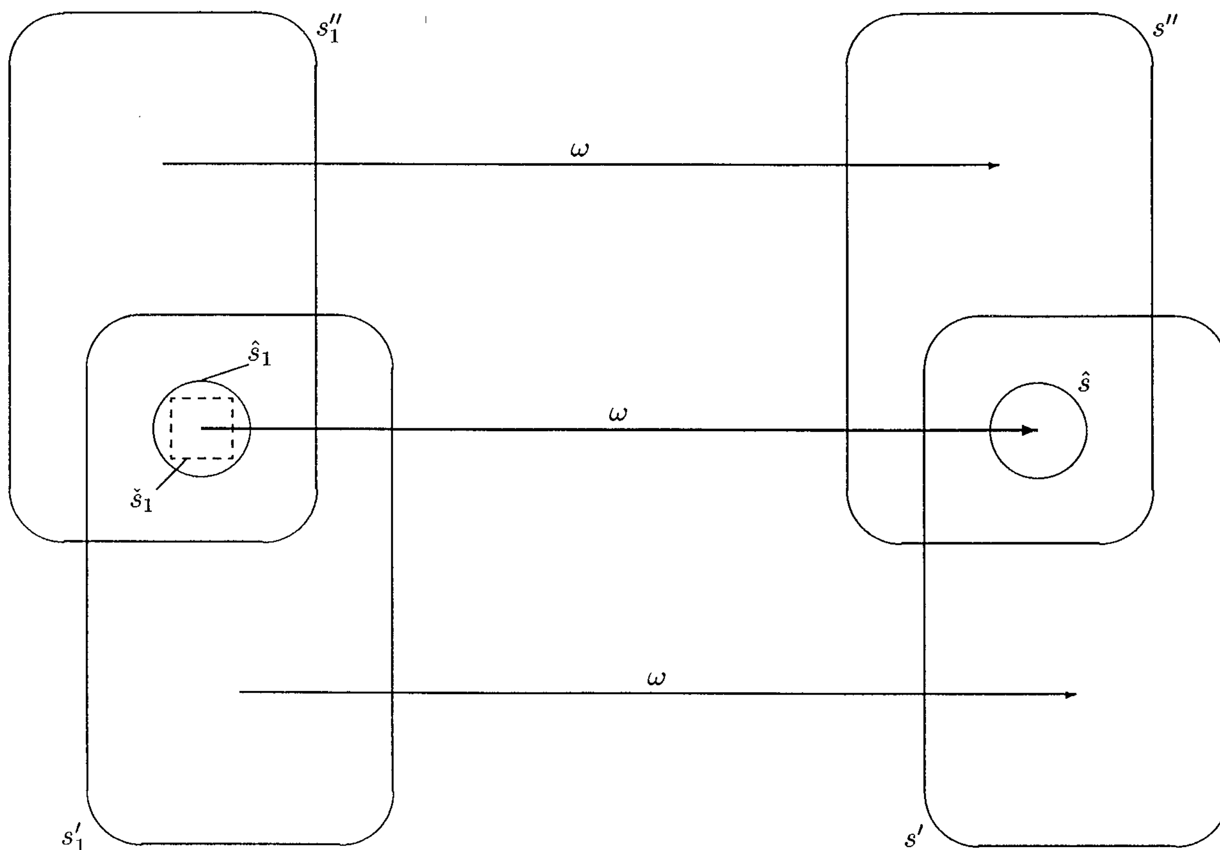
Lemme 0.49 ([85]) *Soit $\Sigma = (S, \leq, \Omega)$ une signature telle que "$\leq$" est un ordre bien fondé. Alors Σ est régulière si et seulement si :*

$$(\forall s'_1, \ldots, s'_n, s', s''_1, \ldots, s''_n, s'' \in S \text{ tels que } (\exists \check{s}_1, \ldots, \check{s}_n \in S), \bigwedge_{1\leqslant i\leqslant n} \left(\check{s}_i \begin{smallmatrix} \leq s'_i \\ \leq s''_i \end{smallmatrix}\right)), \text{ alors :}$$

$$(\forall \omega \in \Omega_{s'_1\times\cdots\times s'_n \to s'} \cap \Omega_{s''_1\times\cdots\times s''_n \to s''}), (\exists \hat{s}_1, \ldots, \hat{s}_n, \hat{s} \in S), \begin{cases} \bigwedge_{1\leqslant i\leqslant n} \left(\check{s}_i \leq \hat{s}_i \begin{smallmatrix} \leq s'_i \\ \leq s''_i \end{smallmatrix}\right) \\ \hat{s} \begin{smallmatrix} \leq s' \\ \leq s'' \end{smallmatrix} \\ \omega \in \Omega_{\hat{s}_1\times\cdots\times\hat{s}_n \to \hat{s}} \end{cases}$$

Dans le cas où "$\leq$" est bien fondé (donc en particulier si S est fini), le lemme précédent assure que la propriété de régularité est décidable.

Pour un opérateur ω d'arité 1, cette propriété de régularité peut se traduire graphiquement comme suit :



Lemme 0.50 ([85]) *Soient Σ une signature régulière et V un ensemble de variables. Tout terme t de $\mathcal{T}_\Sigma(V)$ admet une* **plus petite sorte** *notée $LS(t)$.*

Proposition 0.51 ([85]) *Si Σ est une signature régulière, alors $\mathcal{T}_\Sigma(V)$ est une Σ-algèbre libre sur V.*

Théorème 0.52 ([85]) *Si Σ est une signature régulière, alors $\mathcal{T}_\Sigma$ est une Σ-algèbre initiale dans la catégorie* $\mathbf{OSA}_\Sigma$.

L'hypothèse de régularité est nécessaire pour prouver l'initialité de $\mathcal{T}_\Sigma$: étant donné une signature Σ quelconque, la catégorie $\mathbf{OSA}_\Sigma$ admet toujours un objet initial, mais cet objet ne coïncide pas nécessairement avec $\mathcal{T}_\Sigma$ [85].

0.1.2.3 Équations et règles de réécriture

Dans le cas des sortes ordonnées, il est nécessaire de poser une hypothèse sur la signature, afin que la satisfaction d'une équation soit fermée par isomorphisme. C'est ce qui justifie l'introduction des signatures cohérentes.

Définition 0.53 *Soit $\Sigma = (S, \leq, \Omega)$ une signature. La fermeture symétrique et transitive de "$\leq$" induit une relation d'équivalence dont les classes sont appelées* **composantes connexes** *de $(S, \leq)$.*

Définition 0.54 *Une signature $\Sigma = (S, \leq, \Omega)$ est dite* **cohérente** *si et seulement si elle est régulière et si chaque composante connexe de $(S, \leq)$ admet un élément maximum.*

Remarque En pratique, cette hypothèse n'est évidemment pas contraignante puisqu'il est toujours possible d'étendre une signature régulière Σ en une signature cohérente Σ' par ajout d'un élément maximum $\hat{s}_C$ à l'intérieur de chaque composante connexe C. On a alors $\mathcal{T}_{\Sigma'_{\hat{s}_C}} = \bigcup_{s \in C} \mathcal{T}_{\Sigma_s}$ et $\mathcal{T}_{\Sigma'} = \mathcal{T}_{\Sigma}$.

Définition 0.55 *Soient $\Sigma = (S, \leq, \Omega)$ une signature cohérente et V un ensemble de variables. Une* **Σ-équation** *est une paire $(t_1 == t_2)$ avec $t_1, t_2 \in \mathcal{T}_\Sigma(V)$, tels que $LS(t_1)$ et $LS(t_2)$ soient dans la même composante connexe de $(S, \leq)$.*

Pour les mêmes raisons que dans le cas multi-sorte, les équations sont orientées en règles de réécriture, en vue de rendre les spécifications exécutables.

Définition 0.56 *Une* **règle de réécriture** *est un couple de termes de la forme $(l \longrightarrow r)$, avec $l, r \in \mathcal{T}_\Sigma(V)$, tels que $LS(l)$ et $LS(r)$ soient dans la même composante connexe de $(S, \leq)$, et tels que $var(l) \supseteq var(r)$.*

Définition 0.57 *Soient Σ une signature cohérente et V un ensemble de variables. Un terme t_1 se réécrit en le terme t_2 suivant la règle $(l \longrightarrow r)$ si et seulement si :*

- $(\exists \tau \in occ(t_1)), l \overset{\sigma}{\lesssim} t_1/\tau$,
- $(\exists s \in S), \sigma(l) \in \mathcal{T}_{\Sigma_s}(V) \wedge \sigma(r) \in \mathcal{T}_{\Sigma_s}(V)$,
- *soit $x \in V_s \setminus var(t_1)$, $t_1[\tau \leftarrow x]$ est un terme bien formé,*
- $t_2 = t_1[\tau \leftarrow \sigma(r)]$.

("$\setminus$" désigne la différence ensembliste.)

De même que dans le cas multi-sorte, l'extension à $\mathcal{T}_\Sigma(V)$ de substitutions définies sur V s'effectue par la propriété d'algèbre libre de $\mathcal{T}_\Sigma(V)$. Remarquons que pour toute substitution σ :

$$(\forall x \in V), LS(\sigma(x)) \leq LS(x)$$

Les conditions supplémentaires par rapport à la définition de la réécriture "classique" ont pour objet de refuser l'application de substitutions conduisant à des termes erronés, soit parce que $\sigma(l)$ et $\sigma(r)$ sont de sortes différentes sans borne supérieure commune, soit parce que le remplacement à l'intérieur de t_1 conduirait à un terme mal formé.

Note OBJ3 permet la **réécriture équationnelle**, c'est-à-dire que le filtrage est effectué *modulo* la plus petite congruence engendrée par un ensemble d'équations[12]. Étant donné que nous n'utiliserons pas cette possibilité, nous ne la détaillons pas ici et renvoyons le lecteur intéressé à [116, 52] pour un cadre général, et à [126, 88] pour l'application à OBJ3.

La notion d'équation conditionnelle se généralise facilement au cas des sortes ordonnées, d'après les principes vus au §0.1.1.5. En particulier, la plus petite congruence contenant un ensemble d'équations conditionnelles peut être construite selon la méthode de la proposition 0.38. Les définitions de système de réécriture confluent et possédant la propriété de Church-Rosser sont analogues à celles du §0.1.1.4. Toutefois, l'équivalence de ces deux notions requiert une hypothèse supplémentaire.

[12]En pratique, OBJ3 fournit la réécriture *modulo* les propriétés d'idempotence, d'associativité, de commutativité, et d'existence d'un élément neutre — cf. exemple 0.66.

Définition 0.58 *Un système de réécriture* $\mathcal{R}$ **décroît les sortes**[13] *si et seulement si :*

$$(\forall t_1, t_2 \in \mathcal{T}_\Sigma(V)), t_1 \xrightarrow{*}_{\mathcal{R}} t_2 \Longrightarrow LS(t_1) \geq LS(t_2)$$

Définition 0.59 *Une règle de réécriture* $(l \longrightarrow r)$ *décroît les sortes si et seulement si pour toute substitution* σ *définie sur* $var(l)$ *:*

$$LS(\sigma(l)) \geq LS(\sigma(r))$$

Théorème 0.60 *Un système de réécriture décroît les sortes si et seulement si toutes ses règles décroissent les sortes.*

Note Cette propriété est donc décidable lorsque l'ensemble des sortes est fini.

Théorème 0.61 ([126]) *Soit* $\mathcal{R}$ *un système de réécriture qui décroît les sortes. Les deux propriétés suivantes sont équivalentes :*

- $\mathcal{R}$ *est confluent,*
- $\mathcal{R}$ *possède la propriété de Church-Rosser.*

Dans le cadre des algèbres avec sortes ordonnées, la validité d'une équation est une notion analogue au cas multi-sorte.

Définition 0.62 *Soient* Σ *une signature cohérente et* A *une* Σ*-algèbre. La* Σ*-équation* $(t_1 == t_2)$ *est* **valide** *dans* A *si et seulement si pour toute assignation* $\nu : var(t_1) \cup var(t_2) \rightarrow A$*, alors :*

$$\nu(t_1) = \nu(t_2)$$

Théorème 0.63 ([85]) *Soient* Σ *une signature cohérente,* E *un ensemble de* Σ*-équations, et* V *un ensemble de variables.* $\mathcal{T}_{\Sigma/\equiv_E}(V)$ *est une algèbre libre sur* V*.*

Théorème 0.64 ([85]) *Soient* Σ *une signature cohérente, et* E *un ensemble de* Σ*-équations. Les* Σ*-algèbres qui satisfont* E *et les* Σ*-homomorphismes forment les objets et les flèches d'une catégorie notée* $\mathbf{OSA}_{\Sigma/\equiv_E}$*.* $\mathcal{T}_{\Sigma/\equiv_E}$ *est une algèbre initiale de* $\mathbf{OSA}_{\Sigma/\equiv_E}$*.*

La définition 0.44 des Σ-algèbres, reprise de [79, 84, 126] utilise des fonctions surchargées, partiellement ordonnées par l'inclusion de leurs graphes. Elle inclut la surcharge d'opérateurs, mais sous des conditions très restrictives. En effet, l'hypothèse de régularité implique que les codomaines de deux opérateurs de même nom et de même domaine ne peuvent pas être disjoints, car alors, la condition du lemme 0.49 devient :

$$(\forall s_1, \ldots, s_n, s', s'' \in S), (\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \rightarrow s'} \cap \Omega_{s_1 \times \cdots \times s_n \rightarrow s''}),$$
$$(\exists \hat{s}_1, \ldots, \hat{s}_n, \hat{s} \in S), \bigwedge_{1 \leqslant i \leqslant n} (\hat{s}_i \leq s_i) \wedge \begin{pmatrix} & \leq s' \\ \hat{s} & \\ & \leq s'' \end{pmatrix} \wedge \omega \in \Omega_{\hat{s}_1 \times \cdots \times \hat{s}_n \rightarrow \hat{s}}$$

Pour toute Σ-algèbre :

$$A_{\hat{s}} \begin{matrix} \subseteq A_{s'} \\ \subseteq A_{s''} \end{matrix} \quad \text{d'où : } A_{s'} \cap A_{s''} = \varnothing \Longrightarrow A_{\hat{s}} = \varnothing$$

Par conséquent, la donnée de la plus petite sorte de chaque argument détermine de façon univoque l'exemplaire de l'opérateur utilisé et donc son codomaine. Cette notion est formalisée dans [126] comme suit :

[13] Traduction proposée pour *"a term rewriting system is sort-decreasing"*.

Définition 0.65 ([126]) *Soient Σ une signature et V un ensemble de variables. La* **plus fine levée d'ambiguïtés**[14] *du terme $t \in \mathcal{T}_\Sigma(V)$, notée $LD(t)$, est définie comme suit :*

$$\begin{array}{rcll} \overline{LD(t)} & = & t & (t \in V) \vee (t = \overline{\omega}) \\ LD(\overline{\omega(t_1, \ldots, t_n)}) & = & \overline{\omega_{\check{s}_1 \times \cdots \times \check{s}_n \rightarrow \check{s}}(LD(t_1), \ldots, LD(t_n))} & \end{array}$$

$$\omega : \check{s}_1 \times \cdots \times \check{s}_n \rightarrow \check{s} \text{ étant le plus petit opérateur tel que } \bigwedge_{1 \leqslant i \leqslant n} (LS(t_i) \leq \check{s}_i)$$

Exemple 0.66 *Spéfications de $\mathbb{N}$ muni de l'addition, puis de $\mathbb{Z}$ muni de l'addition — la spécification complète des divers ensembles de nombres ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}[i]$) peut être trouvée dans [79, 126, 85, 88].*

(La directive `protecting NAT` *à l'intérieur de l'objet* `INT` *signifie que les relations entre les termes dont la sorte est définie dans l'objet* `NAT` *ne sont pas perturbées dans* `INT` *— cf. définition précise au §0.1.4.3.)*

```
obj NAT is              ---  Les notations sont celles d'OBJ3.
   sorts Nat NzNat .
   subsorts NzNat < Nat .
   op   0   :           -> Nat .
   op   s_  : Nat       -> NzNat .
   op   _+_ : Nat Nat -> Nat [assoc comm] .
   vars i j : Nat .
   eq       0 + j = j .
   eq   (s i) + j = s(i + j) .
jbo

obj INT is
   protecting NAT .
   sorts Int NzInt .
   subsorts Nat   < Int .
   subsorts NzNat < NzInt < Int .
   op   -_  : NzInt      -> NzInt .
   op   -_  : Int        -> Int .
   op   _+_ : Int   Int -> Int [assoc comm] .
   vars i j : Nat .
   var  p   : Int .
   eq              - 0 = 0 .
   eq             - - p = p .
   eq             p + 0 = p .
   eq   (- s i) + (s j) = (- i) + j .
   eq   (- s i) + (- j) = - s(i + j) .
jbo
```

Deux précisions :

1. Le caractère "_" dans une déclaration d'opérateur représente la place d'un argument. Ainsi, le schéma conditionnel s'exprime dans ce formalisme par "`if_then_else_fi`".

[14] Traduction proposée pour *lowest disambiguation.*

2. Un terme de la forme "`(s x) + (- s y)`", avec `x` et `y` de sorte *Nat*, n'est pas directement filtrable par un membre gauche d'équation, c'est la déclaration de commutativité pour l'addition des entiers relatifs qui permet de le traiter par réécriture équationnelle.

0.1.2.4 Rétractions

Reprenons l'exemple 0.66 et ajoutons-y la définition de l'opérateur de factorielle :

```
_! : Nat -> Nat
```

Si l'on s'en tient aux règles énoncées précédemment, le terme "`(- - 1)!`" est mal formé : en effet, le sous-terme "`- - 1`" admet comme sortes `NzInt` et `Int`, alors que le domaine de "`!`" est `Nat`. Par contre, l'évaluation de "`- - 1`" rend bien un terme admettant `Nat` comme sorte. Dès lors, on comprend l'idée de ne pas considérer uniquement les termes bien formés, mais de passer aux termes *potentiellement bien formés* (selon la terminologie de [79]). Pour cela, on étend une signature $\Sigma = (S, \Omega)$ en une signature $\Sigma^\bullet = (S, \Omega^\bullet)$, $\Omega^\bullet$ étant obtenu par l'ajout à Ω de rétractions[15]. L'ensemble E de Σ-équations est étendu en un ensemble $E^\bullet$. Pour toute sorte s d'une composante connexe C, excepté la sorte maximum ($\max C$), on introduit la **rétraction vers** s par la déclaration :

$$r_{\max C \to s} : \max C \to s$$

et l'équation :

$$r_{\max C \to s}(x) == x$$

x étant une variable de sorte s.

Le terme "`(- - 1)!`" est dès lors accepté et le résultat du *parsing* est :

$$(r_{Int \to Nat}(\texttt{- - 1}))\texttt{!}$$

"$r_{Int\to Nat}$`(1)`" se réduit en `1`, puis "`1 !`" se réduit à son tour en `1`. Par contre, le terme "`(- 1) !`" est *parsé* suivant le même principe et le résultat est :

$$(r_{Int \to Nat}(\texttt{- 1}))\texttt{!}$$

qui est irréductible. Dans ce cas, l'opérateur de rétraction présent dans le résultat peut être compris comme un message d'erreur.

Théorème 0.67 ([85]) *Soient V et W deux ensembles de variables tels que $V \subseteq W$. Si l'extension $\mathcal{T}_{\Sigma/\equiv_E}(V) \to \mathcal{T}_{\Sigma/\equiv_E}(W)$ de l'assignation $V \to \mathcal{T}_{\Sigma/\equiv_E}(W)$ est injective, alors la propriété suivante est vérifiée :*

$$(\forall t_1, t_2 \in \mathcal{T}_{\Sigma/\equiv_E}(V)), t_1 \equiv_E t_2 \Longleftrightarrow t_1 \equiv_{E^\bullet} t_2$$

Dans ce cas, l'ajout des rétractions ne perturbe pas la présentation initiale. Notons que si la signature est complètement habitée, l'hypothèse est vérifiée.

[15]Voir à l'annexe A la définition A.11 de cette notion dans le cadre de la théorie des catégories.

0.1.2.5 Contraintes de sortes

Les déclarations multiples d'un même opérateur, dont la sémantique repose sur la notion de prolongement sont insuffisantes pour décrire toutes les inclusions entre types. Il est nécessaire, parfois, de déclarer qu'un sous-ensemble des termes d'une sorte s est de sorte $\check{s}$, avec $\check{s} \leq s$. Une telle déclaration est une contrainte de sorte.

Définition 0.68 *Soient $\Sigma = (S, \leq, \Omega)$ une signature, Π une famille indicée par S^* de symboles de prédicats, et V un ensemble de variables. Soient $\check{s}, s \in S$, avec $\check{s} \leq s$, $t \in \mathcal{T}_{\Sigma_s}(V)$ et $P \in At_\Sigma(V)$. Une* **contrainte de sorte** *est un triplet de la forme :*

$$(as\ \check{s} : t\ if\ P)$$

P est la **prémisse** *de la contrainte de sorte.*

Exemple 0.69 *Entiers naturels et entiers naturels pairs.*

```
obj NAT-EVEN is
   protecting BOOL .
   sorts Nat Even .
   subsorts Even < Nat .
   op     zero  :        -> Even .
   op     s_    : Nat -> Nat .
   op     even? : Nat -> Bool .
   var    i     : Nat .
   op-as  s     : Nat -> Even for s i if even?(i) .
   eq       even?(zero) = true .
   eq     even?(s zero) = false .
   eq      even?(s s i) = even?(i) .
jbo
```

Définition 0.70 *Soient Σ une signature et A une Σ-algèbre. La contrainte de sorte $(as\ \check{s} : t\ if\ P)$ est* **valide** *dans A si et seulement si pour toute assignation $\nu : var(t) \cup var(P) \to A$, alors :*

$$(A \models \nu(P)) \Longrightarrow \nu(t) \in A_{\check{s}}$$

De même que pour les équations conditionnelles, des restrictions sont introduites sur les prémisses et, en particulier, on y supprime la possibilité de variables à sens existentiel.

Définition 0.71 *Soient $\Sigma = (S, \leq, \Omega)$ une signature cohérente, E un ensemble d'équations conditionnelles, et C un ensemble de contraintes de sortes. Les Σ-algèbres et les Σ-homomorphismes qui satisfont E et C forment les objets et les flèches d'une catégorie notée $\mathbf{OSA}_{\Sigma/\equiv_E/C}$.*

Les contraintes de sortes sont traitées par *extension* de la présentation. Soit la contrainte de sorte :

$$(\text{as}\ \check{s} : t\ \text{if}\ c)$$

On *ordonne* $var(t)$ en une famille $(x_1, \ldots, x_n)$. Soient $s_i = sort(x_i)$, pour $1 \leqslant i \leqslant n$, on associe alors à cette contrainte :

- l'opérateur $f : s_1 \times \cdots \times s_n \to \check{s}$ ($f \notin \Omega$),
- l'équation conditionnelle $(\overline{f(x_1, \ldots, x_n)} == t \text{ if } c)$.

Soient Ω_C et E_C les déclarations et équations conditionnelles associées aux contraintes de sortes de l'ensemble C. Nous avons ainsi défini une nouvelle signature $\tilde{\Sigma} = (S, \Omega \cup \Omega_C)$. L'algèbre initiale de $\mathbf{OSA}_{\Sigma/\equiv_E/C}$ se déduit de celle de $\mathbf{OSA}_{\tilde{\Sigma}/\equiv_{(E \cup E_C)}}$ par oubli des opérateurs de Ω_C.

Théorème 0.72 *Soient $\Sigma = (S, \leq, \Omega)$ une signature cohérente, E un ensemble d'équations conditionnelles, et C un ensemble de contraintes de sortes. Soit $\tilde{\Sigma} = (S, \leq, \Omega \cup \Omega_C)$, et soit U_C le foncteur d'oubli*[16] $\mathbf{OSA}_{\tilde{\Sigma}/\equiv_{(E \cup E_C)}} \to \mathbf{OSA}_{\Sigma/\equiv_E/C}$. *Alors :*

- *U_C est un isomorphisme de catégories,*
- *$\mathcal{T}_{\Sigma/\equiv_E/C} \stackrel{\text{déf}}{=} U_C(\mathcal{T}_{\tilde{\Sigma}/\equiv_{(E \cup E_C)}})$ est une algèbre initiale de* $\mathbf{OSA}_{\Sigma/\equiv_E/C}$.

Les contraintes de sortes sont une généralisation des *déclarations de termes* [69] — on peut comprendre une déclaration de terme comme une contrainte de sorte avec une prémisse trivialement vraie. Définir des *sous-sortes* au moyen de prédicats est également un cas particulier des contraintes de sortes : c'est l'approche d'EQLOG [83]. Elle a été proposée dans [55] pour LPG.

Une approche plus fine a été proposée par Manfred Schmidt-Schauß [158]. Elle consiste à considérer une signature sans sortes (*unsorted*), comprenant des opérateurs, des variables, et des prédicats, puis à considérer l'attribution d'une sorte à un terme comme une fonction (*sort-assignment*).

0.1.2.6 Une autre approche : sémantique de Gert Smolka

La sémantique précédente est une extension très naturelle de l'approche par algèbres multi-sortes, mais elle possède deux inconvénients. D'une part, elle assigne une interprétation à chaque déclaration d'opérateur avec un profil différent, sans regroupement global des fonctions ordonnées par l'inclusion de leurs graphes, d'autre part, l'hypothèse de régularité est nécessaire pour que l'algèbre initiale d'une signature Σ coïncide avec $\mathcal{T}_\Sigma$. Une autre approche [168] — qui se retrouve dans [138] — considère une fonction partielle pour toutes les déclarations concernant le même opérateur. (Dans [168], toutes les déclarations d'un même opérateur admettent obligatoirement la même arité. Nous nous plaçons dans cette hypothèse pour les définitions suivantes.)

Définition 0.73 (Seconde définition d'une Σ-algèbre) *Soit $\Sigma = (S, \leq, \Omega)$ une signature. Une* **Σ-algèbre** *A est constituée de deux familles (A_S, A_Ω) telles que :*

(i) $(\forall s_1, s_2 \in S), s_1 \leq s_2 \Longrightarrow A_{s_1} \subseteq A_{s_2}$

(ii) $(\forall \omega \in \Omega), A_\omega$ *est une fonction partielle de profil* $C_A^{\text{arity}(\omega)} \to C_A$*, où C_A est le* **support** *de A* $(C_A = \bigcup_{s \in S} A_s)$

(iii) $(\forall s_1, \ldots, s_n, s \in S), (\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}), (\forall x_i \in A_{s_i}, 1 \leqslant i \leqslant n), \begin{cases} (x_1, \ldots, x_n) \in \text{dom}(A_\omega), \\ A_\omega(x_1, \ldots, x_n) \in A_s \end{cases}$

[16] Cf. annexe A.

Définition 0.74 (Seconde définition d'un Σ-homomorphisme) *Soit $\Sigma = (S, \leq, \Omega)$ une signature et soient A et B deux Σ-algèbres. Un* **Σ-homomorphisme** *$\gamma : A \rightarrow B$ est la donnée d'une application (également notée γ) $C_A \rightarrow C_B$ telle que :*

(i) $(\forall s \in S), \gamma(A_s) \subseteq B_s$

(ii) $(\forall \omega \in \Omega), \gamma(\text{dom}(A_\omega)) \subseteq \text{dom}(B_\omega)$

(iii) $(\forall \omega \in \Omega, \text{arity}(\omega) = 0), \gamma(A_\omega) = B_\omega$

(iv) $(\forall \omega \in \Omega, \text{arity}(\omega) = n > 0), (\forall (x_1, \ldots, x_n) \in \text{dom}(A_\omega)),$
$$\gamma(A_\omega(x_1, \ldots, x_n)) = B_\omega(\gamma(x_1), \ldots, \gamma(x_n))$$

Étant donné $\Sigma = (S, \leq, \Omega)$ une signature et V un ensemble de variables, [168] introduit l'algèbre des termes engendrée par V d'une façon analogue à la définition 0.47. Pour tout opérateur $\omega \in \Omega$:

$$\text{dom}(\omega) = \{(t_1, \ldots, t_n) \mid \overline{\omega(t_1, \ldots, t_n)} \in \mathcal{T}_\Sigma\}$$

Théorème 0.75 ([168]) *Soit $\Sigma = (S, \leq, \Omega)$ une signature. Les Σ-algèbres et les Σ-homomorphismes forment les objets et les flèches d'une catégorie dont l'objet initial est $\mathcal{T}_\Sigma$.*

Les notions d'assignation, de satisfaction d'équations, de plus petite congruence engendrée par un ensemble d'équations sont analogues aux précédentes, moins l'hypothèse de la signature cohérente.

Théorème 0.76 ([168]) *Soit $\Sigma = (S, \leq, \Omega)$ une signature et E un ensemble d'équations. Les Σ-algèbres qui satisfont E et les Σ-homomorphismes forment les objets et les flèches d'une catégorie dont l'objet initial est $\mathcal{T}_{\Sigma/\equiv_E}$.*

0.1.2.7 Brève comparaison de ces deux approches

L'approche de Gert Smolka permet de dégager l'existence d'une algèbre initiale sous des hypothèses plus faibles que celles de Joseph Goguen et José Meseguer. Il est à remarquer cependant que la syntaxe ne rend pas toujours compte de certaines contraintes imposées par la sémantique. Nous allons mettre en avant cette différence de démarche en étudiant un exemple de signature non régulière. Considérons la signature $\Sigma = (S, \leq, \Omega)$ suivante :

$$\begin{array}{rcl} S & = & \{s_1, \ldots, s_n, s', s''\} \\ \leq & = & \text{égalité entre sortes} \\ \Omega_{s_1 \times \cdots \times s_n \rightarrow s'} & \ni & \omega \\ \Omega_{s_1 \times \cdots \times s_n \rightarrow s''} & \ni & \omega \end{array}$$

Pour toute Σ-algèbre A, la condition (iii) de la définition 0.73 donne :

$$(\forall x_i \in A_{s_i}, 1 \leqslant i \leqslant n), \left\{ \begin{array}{l} (x_1, \ldots, x_n) \in \text{dom}(A_\omega) \\ A_\omega(x_1, \ldots, x_n) \in A_{s'} \\ A_\omega(x_1, \ldots, x_n) \in A_{s''} \end{array} \right.$$

d'où l'on tire :

$$A_\omega(A_{s_1} \times \cdots \times A_{s_n}) \subseteq A_{s'} \cap A_{s''}$$

Par conséquent, même si s' et s'' sont incomparables par "$\leq$", il n'existe aucun modèle de Σ dans lequel leurs supports sont disjoints.

En résumé, nous dirons que l'approche de Gert Smolka correspond mieux à l'intuition — une fonction par opérateur —, et elle est plus adaptée lorsqu'on souhaite étudier les algèbres avec sortes ordonnées sans possibilité de surcharge en dehors des opérateurs ordonnés par l'inclusion des graphes. Néanmoins, outre la prise en compte possible de certaines surcharges (moyennant l'hypothèse de régularité), l'approche de Joseph Goguen et José Meseguer possède les avantages de considérer des fonctions totales et d'être plus proche de la notion habituelle d'algèbre hétérogène — en fait, leur sémantique "colle" davantage à la syntaxe. Dans le cas multi-sorte ("$\leq$" étant dans ce cas l'égalité entre sortes), on retrouve exactement le formalisme du §0.1.1. Au chapitre 4, nous utiliserons la première approche, que nous pensons plus adaptée à notre étude, mais nous verrons que nous aurions pu tout aussi bien choisir la seconde.

Avant l'étude de cette généralisation, nous avions posé au §0.1.1 les bases des concepts théoriques sur lesquels s'appuie la partie fonctionnelle de FP2. Voyons à présent comment ils sont mis en œuvre.

0.1.3 Présentations structurées

Afin de rendre *exécutables* les présentations du langage, et ainsi de pouvoir fournir un évaluateur des termes fermés, il est nécessaire, d'une part, d'orienter les équations, et, d'autre part, d'introduire une certaine hiérarchie entre les opérateurs. La démarche suivie est de partager les opérateurs en **constructeurs** et **non constructeurs** [91], de telle sorte que les non constructeurs puissent être définis par rapport aux constructeurs. Introduisons d'abord la notion de définition complète.

Définition 0.77 *Soient une présentation (S, Ω, E), et $\Gamma \subseteq \Omega$. Un opérateur $\omega \in \Omega_{s_1 \times \cdots \times s_n \to s}$ est* **complètement défini** *par rapport à Γ si et seulement si :*

$$(\forall t_i \in T_{\Sigma_{s_i}}, 1 \leqslant i \leqslant n), (\exists t \in T_{\Sigma_\Gamma}), \overline{\omega(t_1, \ldots, t_n)} \equiv_E t$$

avec $\Sigma_\Gamma = (S, \Gamma)$.

Note "$\Gamma \subseteq \Omega$" est un abus de langage qui exprime que :

$$\begin{array}{rrcl} (\forall s \in S), & \Gamma_{-\to s} & \subseteq & \Omega_{-\to s} \\ (\forall s_1, \ldots, s_n, s \in S), & \Gamma_{s_1 \times \cdots \times s_n \to s} & \subseteq & \Omega_{s_1 \times \cdots \times s_n \to s} \end{array}$$

Suivant cette démarche, les constructeurs constituent les *générateurs* de l'algèbre des termes du type et les autres opérateurs sont complètement définis par rapport aux constructeurs, au moyen de règles de réécriture. Les qualités que doit posséder une présentation pour être exécutable ont été étudiées en [22, 150]. Nous rappelons ci-dessous la définition des présentations structurées.

Définition 0.78 ([150]) *Une présentation (S, Ω, E) est* **structurée** *si :*

(i) toutes les équations de E peuvent être orientées *en règles de réécriture et le système $\mathcal{R}$ obtenu est nœthérien ;*

(ii) il existe $\Gamma \subseteq \Omega$ et $\mathcal{R}_\Gamma \subseteq \mathcal{R}$ — les signatures étant notées $\Sigma = (S, \Omega)$ et $\Sigma_\Gamma = (S, \Gamma)$, et soit V un ensemble de variables — tels que :

(ii-a) les termes des règles de $\mathcal{R}_\Gamma$ sont des termes de $T_{\Sigma_\Gamma}(V)$, et $\mathcal{R}_\Gamma$ est confluent,

(ii-b) $(\forall (l \longrightarrow r) \in \mathcal{R} \setminus \mathcal{R}_\Gamma), l \in T_\Sigma(V) \setminus T_{\Sigma_\Gamma}(V)$

(ii-c) $(\forall t \in \mathcal{T}_\Sigma)$, *toute forme normale de t appartient à* $\mathcal{T}_{\Sigma_\Gamma}$

Les éléments de Γ sont appelés opérateurs **constructeurs.**

D'après les conditions (i) — $\mathcal{R}$ nœthérien — et (ii-*a*), $\mathcal{R}_\Gamma$ est convergent. Par suite, tout terme de $\mathcal{T}_{\Sigma_\Gamma}(V)$ admet une forme normale. La condition (ii-*b*) entraîne que de telles formes normales ne peuvent être réduites par $\mathcal{R}\setminus\mathcal{R}_\Gamma$. Quant à la condition (ii-*c*), elle indique que toute forme normale d'un terme fermé est dans $\mathcal{T}_{\Sigma_\Gamma}$. Par conséquent, tout opérateur, constructeur ou non, est complètement défini par rapport aux constructeurs.

Les conventions de FP2 sont encore un peu plus strictes :

- pas de relations entre opérateurs constructeurs ($\mathcal{R}_\Gamma = \emptyset$) ;
- la racine de la partie gauche de toute règle de réécriture est un opérateur non constructeur, et la racine d'un sous-terme propre[17] de la partie gauche est soit une variable soit un constructeur.

Remarque La convention (ii-*c*) implique que les opérateurs soient totaux. Ce point sera bien sûr repris au chapitre 4, consacré aux traitement des opérateurs partiels

0.1.4 Importation de définitions

0.1.4.1 Notions générales

L'algèbre des termes d'une présentation $\mathcal{P}$ peut subir deux perturbations en cas d'importation de $\mathcal{P}$ dans une présentation $\mathcal{P}'$:

- ajout de termes irréductibles dont la sorte est définie dans $\mathcal{P}$: violation de la propriété *no junk*,
- deux termes différents dans l'algèbre des termes de $\mathcal{P}$ deviennent équivalents dans l'algèbre des termes de $\mathcal{P}'$: violation de la propriété *no confusion.*

Si ces deux propriétés, *no junk* et *no confusion*, sont satisfaites, on dit que $\mathcal{P}$ est **protégée** dans $\mathcal{P}'$ (ou que $\mathcal{P}'$ est une extension **conservative** de $\mathcal{P}$). Nous allons formaliser ces notions. Les définitions suivantes généralisent à toute présentation avec sortes ordonnées la définition 0.39 qui s'applique aux booléens.

Définition 0.79 *Soient* $\mathcal{P} = (\underbrace{S, \leq, \Omega}_{\Sigma}, E)$ *et* $\mathcal{P}' = (\underbrace{S', \leq', \Omega'}_{\Sigma'}, E')$, *telles que :*

(i) $S' \supseteq S$,

(ii) "$\leq'$" *est un prolongement de* "$\leq$",

(iii) $\Omega' \supseteq \Omega$,

(iv) $E' \supseteq E$.

Notons :

$$\mathcal{T}_{\Sigma'/\equiv_{E'}}\restriction S \stackrel{\text{déf}}{=} \{t \mid t \in \mathcal{T}_{\Sigma'/\equiv_{E'}} \wedge sort(t) \in S\}$$

et soit h l'homomorphisme $\mathcal{T}_{\Sigma/\equiv_E} \to \mathcal{T}_{\Sigma'/\equiv_{E'}}\restriction S$.

$\mathcal{P}'$ *est une* **extension** *de* $\mathcal{P}$ *(noté* $\mathcal{P}' \supseteq \mathcal{P}$*) qui est :*

[17] C'est-à-dire un sous-terme dont l'occurrence est différente de λ.

- **complète** *(vérifie la propriété* no junk*) si* h *est surjectif,*
- **consistante** *(vérifie la propriété* no confusion*) si* h *est injectif,*
- **conservative** *si* h *est bijectif.*

Un **enrichissement** *est une extension conservative qui n'ajoute pas de nouvelles sortes.*

Remarque Le théorème 0.67 affirme que, sous les hypothèses requises, l'ajout de rétractions est une extension consistante.

0.1.4.2 Conventions de FP2

FP2 possède deux sortes de **modules** pour définir de telles présentations : les modules de **types** et les modules d'**enrichissements**. Chaque module est considéré comme une extension des modules précédents, c'est-à-dire que lorsqu'on spécifie un nouveau module, les sortes et les opérateurs définis dans les modules de types et d'enrichissements précédents sont accessibles — nous donnons précisément les règles de portée au §0.1.6, après l'introduction informelle à la généricité.

Exemple 0.80 *Le type Bool des booléens.*

```
type Bool
  cons  true, false     : –                 -- Nous ne précisons pas le codomaine des
                                            -- constructeurs, qui est implicite.
  opns  not             : Bool         →  Bool
        and, or, ⇒, ⇔   : Bool × Bool  →  Bool
  vars  b, b0           : Bool
  rules
     <>      not true   ==>  false                  -- "<>" est un séparateur syntaxique
     <>     not false   ==>  true                   -- marquant le début d'une règle de
     <>    true and b   ==>  b                      -- réécriture.
     <>   false and b   ==>  false
     <>     true or b   ==>  true                   -- Noter que les opérateurs d'arité 2
     <>    false or b   ==>  b                      -- peuvent être infixés.
     <>        b ⇒ b0   ==>  (not b) or b0
     <>        b ⇔ b0   ==>  (b ⇒ b0) and (b0 ⇒ b)
endtype
```

Évaluer un terme FP2 consiste à le réécrire, en utilisant une stratégie *leftmost-innermost*, c'est-à-dire par un mécanisme analogue à l'appel par valeur, et en traitant de gauche à droite les divers sous-termes, jusqu'à ce que le terme ne contienne plus que des opérateurs constructeurs : on obtient ainsi sa forme normale. Ainsi :

$$
\begin{aligned}
(\mathit{false} \Rightarrow \mathit{true})\ \mathit{and}\ (\mathit{true}\ \mathit{or}\ \mathit{false}) &\ ==>\ ((\mathit{not}\ \mathit{false})\ \mathit{or}\ \mathit{true})\ \mathit{and}\ (\mathit{true}\ \mathit{or}\ \mathit{false}) \\
&\ ==>\ (\mathit{true}\ \mathit{or}\ \mathit{true})\ \mathit{and}\ (\mathit{true}\ \mathit{or}\ \mathit{false}) \\
&\ ==>\ \mathit{true}\ \mathit{and}\ (\mathit{true}\ \mathit{or}\ \mathit{false}) \\
&\ ==>\ \mathit{true}\ \mathit{and}\ \mathit{true} \\
&\ ==>\ \mathit{true}
\end{aligned}
$$

Exemple 0.81 *Définition du type Nat des entiers naturels, puis d'un enrichissement utilisant les définitions des types Nat et Bool, et spécifiant l'égalité entre entiers.*

type *Nat*
  cons *zero* : –
    succ : *Nat*
endtype

enr
  opns = : $Nat \times Nat \rightarrow Bool$
  vars i, j : *Nat*
  rules
    <> *zero* = *zero* ==> *true*
    <> *zero* = *succ*(*j*) ==> *false*
    <> *succ*(*i*) = *zero* ==> *false*
    <> *succ*(*i*) = *succ*(*j*) ==> $i = j$
endenr

(Il est bien sûr possible d'utiliser la notation décimale pour désigner des entiers.)

0.1.4.3 Directives d'OBJ3 — Comparaison avec FP2

En OBJ3, *aucune* visibilité des objets définis précédemment n'est implicite, contrairement à FP2. Posons $\mathcal{P} = (S, \Omega, E), \mathcal{P}' = (S', \Omega', E')$, avec $\mathcal{P}' \supseteq \mathcal{P}$. Les importations s'expriment à l'aide de trois directives :

- la directive "$\mathcal{P}'$ *is protecting* $\mathcal{P}$" signifie que $\mathcal{P}'$ est une extension conservative de $\mathcal{P}$, et que :

  $\star$ $(\forall s' \in S' \setminus S), (\forall s \in S), s' \nleq s,$

  $$\star \left.\begin{array}{r}(\forall(\omega : s_1' \times \cdots \times s_n' \rightarrow s') \in \Omega' \setminus \Omega),\\ (\forall(\omega : s_1 \times \cdots \times s_n \rightarrow s) \in \Omega),\end{array}\right\} \neg\big(\bigwedge_{1 \leqslant i \leqslant n} (s_i' \leq s_i) \wedge (s' \leq s)\big)$$

  autrement dit, la plus fine levée d'ambiguïtés d'un terme quelconque de $\mathcal{T}_{\mathcal{P}}$ est identique dans $\mathcal{P}$ et dans $\mathcal{P}'$;

- la directive "$\mathcal{P}'$ *is extending* $\mathcal{P}$" signifie que $\mathcal{P}'$ est une extension consistante de $\mathcal{P}$;

- la directive "$\mathcal{P}'$ *is using* $\mathcal{P}$" est la moins restrictive : aucune hypothèse n'est faite sur la protection des définitions de $\mathcal{P}$; du point de vue de l'implantation, les définitions de $\mathcal{P}$ sont *copiées* avant leur utilisation dans $\mathcal{P}'$.

Exemple 0.82 *Spécifications, à partir de l'ensemble* $\mathbb{N}$, *des ensembles* $\mathbb{N} \cup \{\infty\}$ *et* $\mathbb{Z}/2\mathbb{Z}$.

```
obj NAT is
   sorts Nat .
   op   0   :             -> Nat .
   op   s_  : Nat         -> Nat .
   op   _+_ : Nat Nat -> Nat .
   vars i j : Nat .
   eq       0 + j = j .
   eq   (s i) + j = s(i + j) .
jbo
```

```
obj NAT-INF is
   extending NAT .
   op   infinity : -> Nat .
   var  n          : Nat .
   eq      s infinity = infinity .
   eq   infinity + n = infinity .
jbo

obj Z/2Z is
   using NAT.
   var n : Nat .
   eq   s s n = n .
jbo
```

En FP2, l'impossibilité de générer des équations entre constructeurs garantit la propriété de *no confusion*[18]. La propriété *no junk* est assurée à la fois par l'impossibilité d'ajouter des constructeurs à une sorte déjà définie[19] et l'obligation de pouvoir exprimer toute forme normale uniquement en fonction de constructeurs.

Il est certain que la visibilité de tous les objets précédemment définis, comme en FP2, est une approche plutôt orientée vers la programmation, alors que l'approche d'OBJ3, qui vise à fabriquer un graphe de dépendance strict des modules, est peut-être plus adaptée à la *réutilisabilité*. Remarquons toutefois qu'en FP2, la possibilité d'opérateurs *privés* (cachés) permet de spécifier avec précision un découpage interface/corps. (Ces opérateurs privés ne pouvant être constructeurs, ils sont par hypothèse complètement définis par rapport aux constructeurs, et ainsi, la propriété d'existence d'une algèbre initiale n'est pas perturbée par l'ajout de ces opérateurs privés.)

0.1.5 Introduction informelle à la généricité en FP2

L'idée de base de la généricité est de pouvoir paramétrer une présentation par une autre présentation. Comme toute présentation algébrique, la présentation en paramètre est représentée par ses sortes, ses opérateurs, et sa relation de congruence spécifiée au moyen d'équations. Ces sortes et ces opérateurs deviendront des types et des opérateurs **formels** de la présentation paramétrée. La description d'une présentation que l'on désire passer en paramètre se fait à l'aide d'une **propriété**.

Exemple 0.83 *Propriété d'égalité sur un type t muni de l'opérateur eq.*

```
prop Equality[t / eq]
   opns eq      : t × t  →  Bool
   vars x, y, z : t
   eqns
      <>                                 x eq x  ==  true
      <>                                 x eq y  ==  y eq x
      <>  ((x eq y) and (y eq z)) ⇒ (x eq z)  ==  true
endprop
```

$[t\ /\ eq : t \times t \rightarrow Bool]$ est la **signature** de la propriété *Equality*.

Remarque Les équations caractérisant un tel opérateur formel d'égalité n'ont qu'un caractère purement descriptif, ce ne sont *pas* des règles de réécriture utilisées par l'évaluateur.

Exemple 0.84 *L'algèbre de la présentation du type Nat, enrichie de la définition de "=", est un* modèle *de la propriété Equality. Ceci s'exprime par la déclaration de* **modèle** *suivante :*

model *EQNAT* **is** *Equality*[*Nat* / =]
endmodel

[18] Par "générer des équations entre constructeurs", on entend non seulement leur spécification directe par l'utilisateur, mais aussi leur occurrence lors de la construction de la congruence engendrée par les équations. Ainsi la définition suivante de l'opérateur $f : Bool \rightarrow Bool$ (b étant une variable de sorte $Bool$) :

$$f(true) == true$$
$$f(b) == false$$

est considérée comme incorrecte, car elle engendre l'équation ($true == false$).

[19] D'après la définition, un enrichissement ne peut pas introduire de nouveaux opérateurs constructeurs.

Une telle déclaration représente en fait des liaisons entre la signature d'une propriété et la signature d'un de ses modèles, données d'après l'ordre lexical convenu dans l'en-tête de la spécification de la propriété. Ici : $[t \mapsto Nat, eq \mapsto =]$.

Exemple 0.85 *Nous allons définir un opérateur eq_3, d'égalité à trois arguments, sur un type formel admettant un opérateur d'égalité (à deux arguments). On dit que l'opérateur eq_3* **exige** *(*requiert*) la propriété d'égalité.*

```
enr req Equality[t / eq2]
   opns  eq3      :  t × t × t  →  Bool
   vars  x, y, z  :  t
   rules
      <>  eq3(x, y, z)  ==>  (x eq2 y) and (x eq2 z)
endenr
```

eq_3 est un opérateur **générique** qui agit sur trois arguments de sorte t, ses **paramètres formels** sont la *sorte formelle t* et l'*opérateur formel eq_2*, ces paramètres étant introduits au moyen de la propriété *Equality*.

Cet opérateur peut être **instancié** par tout modèle de la propriété *Equality* : les paramètres formels sont alors remplacés par les paramètres effectifs. Désignant une instanciation par ".", nous pouvons construire $eq_3.Equality[Bool\ /\ \Leftrightarrow]$, de profil $Bool \times Bool \times Bool \rightarrow Bool$, et évaluer :

$$
\begin{aligned}
eq_3.Equality[Bool\ /\ \Leftrightarrow](false, true, false) &==> (false \Leftrightarrow true)\ and\ (false \Leftrightarrow false) \\
&==> false
\end{aligned}
$$

Avec une autre instanciation : $eq_3.Equality[Nat\ /\ =](1,1,1) ==> true$

En utilisant la déclaration de l'exemple 0.84 : $eq_3.EQNAT(1,1,1)$.

Remarque Nous verrons au chapitre 3, consacré aux facilités de notations, qu'en fait, les expressions — beaucoup plus légères — $eq_3(false, true, false)$ et $eq_3(1,1,1)$ sont licites et analysées respectivement comme $eq_3.Equality[Bool\ /\ \Leftrightarrow](false, true, false)$ et $eq_3.EQNAT(1,1,1)$.

Exemple 0.86 *Propriété Ftype (*Formal type*) : tout type est sous-jacent à un modèle de cette propriété.*

```
prop Ftype[t / ]
endprop
```

Exemple 0.87 *Type des séquences linéaires construites sur un type formel. La sorte Seq* exige *la propriété Ftype. ("<+" est l'ajout en tête de séquence.)*

```
type Seq req Ftype[t / ]
   cons  nil    :                         −
         <+     :                   t  ×  Seq.Ftype[t / ]
   opns  +      :  Seq.Ftype[t / ]  ×  Seq.Ftype[t / ]  →  Seq.Ftype[t / ]
                                                            -- Concaténation.
   vars  x      :  t
         s, s0  :  Seq.Ftype[t / ]
   rules
      <>         nil + s0  ==>  s0
      <>  (x <+ s) + s0    ==>  x <+ (s + s0)
endtype
```

Une notation plus légère est disponible : $[x_1, \ldots, x_n]$ pour $x_1 <+ (\ldots <+ (x_n <+ nil))$.

Dès lors que tout type est bien évidemment un modèle de *Ftype*, nous pouvons donc construire les séquences d'entiers naturels (*Seq.Ftype*[*Nat* /]), les séquences de booléens (*Seq.Ftype*[*Bool* /]), ...

Si tout type est un modèle de *Ftype*, alors en particulier le type *Seq.Ftype*[*Nat* /]. De proche en proche, nous comprenons qu'il est possible de former :

$$Seq.Ftype[Seq.Ftype[Nat\ /\ ]\ /\ ]$$
$$Seq.Ftype[Seq.Ftype[Seq.Ftype[Nat\ /\ ]\ /\ ]\ \ /\ ]$$
$$\vdots$$

En généralisant, on peut parler de séquence dont le paramètre est une séquence générique, sous la propriété exigée *Ftype*.

Note Pour une instanciation par un modèle de *Ftype*, nous allons utiliser une notation abrégée en ne donnant que la sorte effective : *Seq*[*Nat*], *Seq*[*Bool*], *Seq*[*Seq*[*Nat*]], ... Nous repréciserons ce détail au chapitre 3, lorsque nous étudierons les raccourcis de notations.

À travers l'exemple suivant, nous allons montrer comment on peut reporter de proche en proche une propriété sur un type générique.

Exemple 0.88 *Égalité des séquences génériques.*

```
enr req Equality[t / eq]
  opns  =       :  Seq[t] × Seq[t]  →  Bool
  vars  x1,x2   :  t
        s1,s2   :  Seq[t]
  rules
    <>        nil  =  nil          ==>  true
    <>        nil  =  (x2 <+ s2)   ==>  false
    <>  (x1 <+ s1) =  nil          ==>  false
    <>  (x1 <+ s1) =  (x2 <+ s2)   ==>  if x1 eq x2 then s1 = s2 else false
                                        endif
endenr
```

Remarque Nous étudierons plus en détail la surcharge des opérateurs au chapitre 3. Disons pour l'instant qu'elle est permise. Néanmoins, pour pouvoir citer sans ambiguïté un opérateur particulier, et étant donné que la facilité de surcharge rend certaines expressions ou sous-expressions confuses, nous noterons parfois les divers exemplaires d'un symbole surchargé avec des indices suggérant l'exemplaire utilisé, ainsi "$=_N$" pour l'égalité entre entiers naturels, "$=_S$" pour l'égalité entre séquences. Ce n'est qu'une commodité de notation utilisée dans la thèse, l'utilisateur de FP2 peut les désigner toutes deux par "=".

Si nous appliquons cet opérateur à des séquences d'entiers naturels, l'appel avec spécification des paramètres effectifs est :

$$=_S.Equality[Nat\ /\ =_N]([0],[0]) \quad \text{ou} \quad =_S.EQNAT([0],[0])$$

On peut remarquer que le type "séquences d'entiers naturels", muni de l'opérateur "$=_S$" instancié par le modèle *EQNAT*, est un modèle de l'égalité. Il est donc possible d'appliquer l'opérateur générique "$=_S$" à des séquences dont les éléments dont des séquences d'entiers, en l'instanciant comme suit :

$$=_S.Equality[Seq[Nat]\ /\ =_S.Equality[Nat\ /\ =_N]]([[0]],[[0]]) \tag{0.1}$$

Plus généralement, pour une sorte t, s'il existe un opérateur eq tel que toute algèbre du type t muni de eq soit un modèle de l'égalité, alors toute algèbre du type $Seq[t]$ muni de l'opérateur générique "$=_S$" paramétré par $Equality[t\ /\ eq]$ est un modèle de l'égalité. Ceci est exprimé par le **modèle générique** suivant :

model $EQSEQ$ **req** $Equality[t\ /\ eq]$ is $Equality[Seq[t]\ /\ =_S]$
endmodel

et l'expression (0.1) peut être également décrite par :

$$=_S.EQSEQ.EQNAT([[0]],[[0]])$$

dans laquelle le modèle $EQSEQ$ est instancié par $EQNAT$, pour former un modèle qui instancie "$=_S$".

De façon analogue, on peut écrire :

$$=_S.EQSEQ.EQSEQ.EQNAT([[[0]]],[[[0]]])$$
$$=_S.EQSEQ.EQSEQ.EQSEQ.EQNAT([[[[0]]]],[[[[0]]]])$$
$$\vdots$$

Remarque Ainsi que nous l'avons précédemment signalé à propos de l'exemple 0.85, nous verrons au chapitre 3 que les expressions [0] = [0], [[0]] = [[0]], [[[0]]] = [[[0]]], [[[[0]]]] = [[[[0]]]] sont acceptées par l'analyseur et traitées comme les expressions développées que nous venons de décrire.

0.1.6 Portée des objets en FP2 : tableau récapitulatif

Chaque module peut utiliser toutes les définitions qui le précèdent. Les références "en avant" n'étant pas permises, il n'est pas possible de spécifier des définitions mutuellement récursives de types ou de modèles. Le tableau suivant précise, pour les objets d'un module, si la surcharge est possible, et si ces objets seront visibles après la lecture de ce module.

	Possibilité de surcharge ?	Visibilité ?
Constructeurs	oui	oui
Opérateurs	oui	oui
Opérateurs privés (locaux)	oui	-
Opérateurs d'une propriété	-	-
Opérateurs formels	-	-
Variables	-	-

0.2 Programmation parallèle

Les deux principaux langages qui sont à la base de la programmation parallèle sont CSP et CCS.

Le langage CSP [96] intègre en tant que primitive du langage la programmation de canaux d'entrée-sortie entre deux processus. Au niveau des opérations internes d'un processus, sa sémantique est celle d'un langage algorithmique traditionnel. Le langage OCCAM [137] est issu de CSP.

Dans le formalisme de CCS [140], les transitions des processus, ainsi que les communications entre ces processus, sont décrites au moyen d'une algèbre. Les principaux buts de CCS sont d'étudier les

comportements et de pouvoir effectuer des comparaisons entre processus, c'est-à-dire entre les termes qui les représentent dans cette algèbre, comparaisons plus ou moins fines selon le niveau d'observation considéré. Le langage parallèle LOTOS (Language Of Temporal Ordering Specification) [181, 67], principalement utilisé pour décrire des protocoles, est directement inspiré de CCS.

Alors que CSP a privilégié la programmation, alors que CCS est davantage un formalisme de comparaison entre processus plutôt qu'un langage de programmation, le but de FP2 est d'une part de fournir un langage de spécification et de programmation de processus parallèles communicants, d'autre part de permettre l'analyse des comportements et la comparaison de processus.

0.2.1 Processus

En FP2, l'unité de base du parallélisme est le **processus**. Un processus FP2 s'exécute de manière séquentielle et communique au moyen de messages (termes) avec un environnement formé d'autres processus.

0.2.1.1 Le niveau syntaxique

Nous allons décrire ci-après le niveau syntaxique des processus, en y ajoutant parfois quelques considérations opérationnelles pour faciliter la compréhension. Au même titre que tous les termes de la partie fonctionnelle, *tous* les messages des processus sont typés. Par conséquent, toute spécification de processus va utiliser une présentation algébrique.

Définition 0.89 ([165]) *Soit (S, Ω, E) une présentation algébrique. Une* **signature de processus** *est un couple $\langle \Pi, K \rangle$ où Π est une famille indicée par S^* d'ensembles de* **constructeurs d'états,** *et K une famille indicée par S d'ensembles de* **connecteurs.**

Définition 0.90 ([165]) *Soient Σ une signature algébrique et $\langle \Pi, K \rangle$ une signature de processus. Soit également V un ensemble de variables.*

- *Un* **terme de communication** *est de la forme $\overline{C(t)}$, avec $C \in K_s, s \in S$ et $t \in T_{\Sigma_s}(V)$.*
- *Un* **terme d'événement** *est un ensemble, éventuellement vide, de termes de communication :*

  $$\{\overline{C_1(t_1)}, \ldots, \overline{C_n(t_n)}\}$$

  les noms de connecteurs $(C_i)_{1 \leqslant i \leqslant n}$ étant deux à deux distincts.
- *Un* **terme d'état** *est de la forme $\overline{A(t_1, \ldots, t_n)}$, avec $A \in \Pi_{s_1 \times \cdots \times s_n}, s_1, \ldots, s_n \in S$, et $t_i \in T_{\Sigma_{s_i}}(V)$, pour $1 \leqslant i \leqslant n$. L'ensemble des termes d'état est noté T_Π.*
- *Une* **règle initiale** *est composée d'un terme d'état* fermé, *et est notée :*

  $$(==> \pi)$$

- *Une* **règle de transition** *est un triplet noté comme suit :*

  $$(\pi_1 : e ==> \pi_2)$$

  où π_1 et π_2 sont des termes d'état et e un terme d'événement. π_1 et π_2 sont respectivement la **précondition** *et la* **postcondition** *de la règle de transition.*

Définition 0.91 ([165]) *Une* **présentation de processus** *est un quadruplet* $\langle \Pi, K, TR, I \rangle$ *où* $\langle \Pi, K \rangle$ *est une signature de processus,* TR *un ensemble de règles de transition, et* I *un ensemble non vide de règles initiales.*

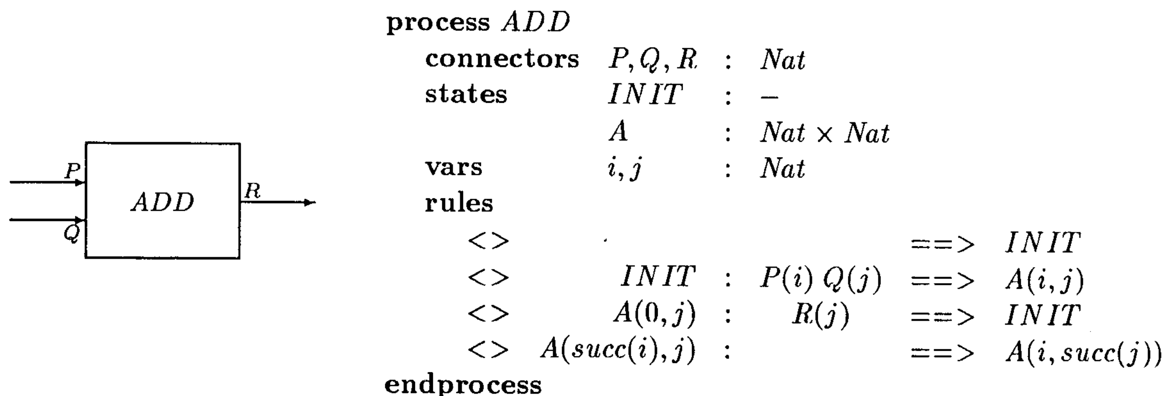
L'ensemble des présentations de processus est noté **PP**.

Les **connecteurs** assurent les communications avec l'environnement du processus. Les constructeurs d'états servent à exprimer les **états possibles** du processus. Une règle initiale spécifie un état initial possible[20]. Une règle de transition décrit une possibilité pour le processus de passer d'un état à un autre.

Au départ d'une exécution, on choisit, de façon **non déterministe**, un état initial parmi les états initiaux possibles. Détaillons maintenant le *modus operandi* d'une transition. Les règles de transition dont la précondition filtre[21] l'état actuel du processus sont dites **pré-applicables**. Si aucune règle de transition n'est pré-applicable, le processus est **terminé**. Sinon, on appelle **applicable** toute règle pré-applicable pour laquelle l'environnement est prêt à contribuer à l'occurrence de l'événement. Lorsqu'on applique une telle règle, l'événement se produit effectivement, et le terme d'état est *réécrit*. Si l'application de règles est possible, l'une d'entre elles est nécessairement appliquée au bout d'un temps fini. Comme en ce qui concerne les règles initiales, le choix entre les diverses règles applicables est non déterministe.

Par conséquent, une **histoire possible** d'un processus FP2 est une séquence de transitions d'états, partant d'un état initial possible, ces transitions étant *étiquetées* par des événements.

Exemple 0.92 *Addition de deux entiers naturels.*




```
process ADD
   connectors  P,Q,R   : Nat
   states      INIT    : -
               A       : Nat x Nat
   vars        i,j     : Nat
   rules
      <>                              ==> INIT
      <>         INIT  : P(i) Q(j)    ==> A(i,j)
      <>       A(0,j)  :   R(j)       ==> INIT
      <> A(succ(i),j)  :              ==> A(i,succ(j))
endprocess
```

P, Q et R sont des connecteurs par lesquels on peut envoyer ou recevoir des termes de sorte *Nat*. Donnons à présent une interprétation opérationnelle de ce processus. Quand deux entiers naturels se présentent aux connecteurs P et Q, leur somme est retournée au bout d'un temps fini par le connecteur R, et le processus ADD est prêt à recevoir deux autres entiers. Un exemple d'histoire possible de ce processus est :

$$INIT \overset{P(2)\,Q(3)}{\Longrightarrow} A(2,3) \Longrightarrow A(1,4) \Longrightarrow A(0,5) \overset{R(5)}{\Longrightarrow} INIT \overset{...}{\Longrightarrow} \ldots$$

Remarque ADD est un processus déterministe.

[20] État initial *possible*, puisqu'une présentation d'un processus FP2 peut admettre *plusieurs* règles initiales.

[21] Le filtrage entre termes d'états est bien sûr une notion semblable à celle de la définition 0.16. Du point de vue opérationnel, FP2 impose dans ce but que les sous-termes d'une précondition soient uniquement composés de variables ou de constructeurs — cette restriction est la même que pour les parties gauches des règles de réécriture de la partie fonctionnelle (cf. §0.1.3).

0.2.1.2 Le niveau sémantique

Au niveau sémantique, un processus FP2 représente un système de transition communicant [123, 146, 115, 160, 165].

Définition 0.93 *Un* **système de transition** *est la donnée d'un quadruplet $\langle Q, L, \rightarrow, \mathcal{I}\rangle$ où Q est un ensemble non vide d'***états***, L un ensemble d'***événements***, "$\rightarrow$" une* **relation de transition** *($\rightarrow \subseteq Q \times L \times Q$), et $\mathcal{I}$ un ensemble non vide d'***états initiaux** *($\mathcal{I} \subseteq Q$).*

On note un élément (q_1, l, q_2) de la relation "$\rightarrow$" par "$q_1 \xrightarrow{l} q_2$".

Définition 0.94 *Soient Val un ensemble de* **valeurs** *et K un ensemble de connecteurs. Un système de transition $\langle Q, L, \rightarrow, \mathcal{I}\rangle$ est un* **système communicant** *si $L = Val^K$, l'ensemble de toutes les fonctions partielles de K à valeurs dans Val. Un tel système est alors noté :*

$$\langle Q, K, Val, \rightarrow, \mathcal{I}\rangle$$

La classe des systèmes communicants est désignée par **CS** *(notation de [165]).*

Toujours suivant [165], nous dirons qu'une présentation de processus $\langle \Pi, K, TR, I\rangle$ opérant sur une présentation algébrique (S, Ω, E), avec $\Sigma = (S, \Omega)$, représente le système communicant :

$$\langle T_\Pi, K, T_{\Sigma/\equiv_E}, \rightarrow_{TR}, I\downarrow\rangle$$

où :

$$\begin{array}{rcl} \rightarrow_{TR} & = & \{\sigma(\pi_1) \xrightarrow{\sigma(e)} \sigma(\pi_2) \mid \sigma \in \mathbf{Subst_g}(T_\Sigma(var(\pi_1) \cup var(e) \cup var(\pi_2))), (\pi_1 : e ==> \pi_2) \in TR\} \\ I\downarrow & = & \{\pi\downarrow \mid (==> \pi) \in I\} \end{array}$$

avec :

$$\begin{array}{rcll} \overline{A}\downarrow & = & \overline{A} & (A \in \Pi_{-}) \\ \overline{A(t_1, \ldots, t_n)}\downarrow & = & \overline{A(t_1\downarrow, \ldots, t_n\downarrow)} & (A \in \Pi_{s_1 \times \cdots \times s_n}) \end{array}$$

Cette approche définit une **sémantique déclarative** $\mathcal{P}[\![.]\!] : \mathbf{PP} \rightarrow \mathbf{CS}$.

Le formalisme des systèmes communicants permet de décrire précisément les opérateurs entre processus que nous allons introduire maintenant. Nous allons nous contenter de les spécifier au niveau des présentations de processus, le lecteur intéressé trouvera en [165] une description complète de ces opérateurs dans le cadre des systèmes communicants.

0.2.2 Opérateurs entre processus

Ainsi que d'autres langages de programmation parallèle, FP2 possède des opérateurs permettant, à partir de processus, de construire des réseaux. Pour l'instant, il est impossible de passer des présentations de processus en paramètres, donc a fortiori impossible pour l'utilisateur de définir lui-même de nouveaux opérateurs entre processus. Ces opérateurs sont par conséquent en nombre fixe. La définition d'un tel opérateur consiste en la donnée de règles permettant d'expliciter le réseau

construit sous forme d'une présentation de processus, c'est-à-dire comme si ce réseau était un processus élémentaire, directement spécifié par l'utilisateur[22]. Nous allons détailler les opérateurs les plus employés en FP2, et dont nous aurons besoin au chapitre 5. Les trois premiers sont issus du calcul présenté en [109].

0.2.2.1 Composition parallèle

Soient P_1 et P_2 deux présentations de processus ne partageant aucun connecteur de même nom. On définit $P_1 \parallel P_2$ comme étant une présentation de processus dont tout terme d'état provient du produit d'un terme d'état de P_1 par un terme d'état de P_2, et dont une transition est soit une transition de P_1 et une transition de P_2 simultanées, soit une transition de P_1 seulement, soit une transition de P_2 seulement. Nous allons spécifier ceci plus formellement. Nous supposons sans perte de généralité que les variables des règles de P_1 et celles des règles de P_2 forment deux ensembles disjoints.

Définissons tout d'abord le produit de deux termes d'état :

$$s_1(t_1, \ldots, t_p) * s_2(u_1, \ldots, u_q) \stackrel{\text{déf}}{=} s_1.s_2(t_1, \ldots, t_p, u_1, \ldots, u_q)$$

(où $s_1.s_2$ est un constructeur d'état obtenu par concaténation du symbole s_1, du caractère "." et du symbole s_2 — le domaine de $s_1.s_2$ est la concaténation des domaines de s_1 et s_2, en comprenant ces domaines comme des *séquences* de sortes),

puis le produit de deux règles initiales :

$$(==> post_1(t_1, \ldots, t_p)) * (==> post_2(u_1, \ldots, u_q)) \stackrel{\text{déf}}{=} (==> post_1.post_2(t_1, \ldots, t_p, u_1, \ldots, u_q))$$

et enfin, le produit de deux règles de transition :

$$(pré_1(t_1, \ldots, t_p) : ev_1 ==> post_1(u_1, \ldots, u_q)) * (pré_2(v_1, \ldots, v_r) : ev_2 ==> post_2(w_1, \ldots, w_s)) \stackrel{\text{déf}}{=}$$
$$(pré_1.pré_2(t_1, \ldots, t_p, v_1, \ldots, v_r) : ev_1 \cup ev_2 ==> post_1.post_2(u_1, \ldots, u_q, w_1, \ldots, w_s))$$

Pour une présentation de processus P quelconque, nous allons noter par $connectors(P)$, $states(P)$, $initial\text{-}rules(P)$, $tr\text{-}rules(P)$, respectivement l'ensemble de ses connecteurs, l'ensemble de ses constructeurs d'états, l'ensemble de ses règles initiales et l'ensemble de ses règles de transition. Pour une règle de transition r, $precondition(r)$, $postcondition(r)$, et $event(r)$ désigneront respectivement la précondition, la postcondition, et l'événement de la règle r.

Afin de pouvoir exprimer "les transitions d'un processus seulement", nous introduisons les "règles immobiles" (*idle rules*) d'une présentation de processus P :

$$idle\text{-}rules(P) = \{(s(x_1, \ldots, x_{\text{arity}(s)}) ==> s(x_1, \ldots, x_{\text{arity}(s)})) \mid s \in states(P)\}$$

[22] Ce procédé, appelé ***mise à plat (flattening)*** dans [165], est justifié par l'existence d'un opérateur équivalent pour les systèmes communicants, tel que cette mise à plat soit correcte. Ainsi, pour l'opérateur "$\parallel$" présenté au §0.2.2.1, il existe un opérateur de mise en parallèle des systèmes communicants ("$\parallel_c$"), tel que :

$$\mathcal{P}[P_1 \parallel P_2] = \mathcal{P}[P_1] \parallel_c \mathcal{P}[P_2]$$

Finalement, nous obtenons :

$$\begin{array}{rcll} connectors(P_1 \parallel P_2) & = & connectors(P_1) \cup connectors(P_2) & \\ states(P_1 \parallel P_2) & = & \{s_1 \,.\, s_2 & \mid\ s_1 \in states(P_1) \wedge s_2 \in states(P_2)\} \\ initial\text{-}rules(P_1 \parallel P_2) & = & \{i_1 * i_2 & \mid\ i_1 \in initial\text{-}rules(P_1) \wedge i_2 \in initial\text{-}rules(P_2)\} \\ tr\text{-}rules(P_1 \parallel P_2) & = & \{r_1 * r_2 & \mid\ (r_1 \in tr\text{-}rules(P_1) \wedge r_2 \in tr\text{-}rules(P_2)) \vee \\ & & & \quad (r_1 \in tr\text{-}rules(P_1) \wedge r_2 \in idle\text{-}rules(P_2)) \vee \\ & & & \quad (r_1 \in idle\text{-}rules(P_1) \wedge r_2 \in tr\text{-}rules(P_2))\} \end{array}$$

Remarque Cet opérateur existe en Meije[23], et il y est défini de manière analogue. Par contre, en CCS, l'opérateur de mise en parallèle (également noté "||") est défini pour deux processus P_1 et P_2 admettant un point de synchronisation, et s'exprime par interfoliage : une transition de $P_1 \parallel P_2$ est soit une transition de P_1 seulement, soit une transition de P_2 seulement, soit un échange de valeurs entre P_1 et P_2 via leur point de synchronisation.

0.2.2.2 Connexion

Soient P une présentation de processus, A et B deux de ses connecteurs, alors $P + A.B$ désigne le processus dans lequel A et B peuvent échanger des valeurs (qui doivent bien sûr être de même sorte). La présentation du processus résultant est obtenue en effectuant l'unification des messages. Ainsi :

$$\begin{array}{rcl} connectors(P + A.B) & = & connectors(P) \\ states(P + A.B) & = & states(P) \\ initial\text{-}rules(P + A.B) & = & initial\text{-}rules(P) \\ tr\text{-}rules(P + A.B) & = & tr\text{-}rules(P) \cup connection\text{-}rules(P, A, B) \end{array}$$

avec :

$$\begin{array}{rcl} connection\text{-}rules(P, A, B) & = & \{\tilde{\sigma}(r) \mid r \in tr\text{-}rules(P), (\exists u, v : A(u), B(v) \in event(r)), u \overset{\sigma}{\approx} v\} \\ \tilde{\sigma}(r) & = & \sigma(precondition(r) : event(r) \setminus \{A(u), B(v)\} ==> postcondition(r)) \end{array}$$

Note La formulation de *connection-rules*(P, A, B) que nous venons de donner n'est valable que lorsque l'unification de deux termes est décidable statiquement, ce qui nous restreint aux termes composés uniquement de variables et de constructeurs[24] : c'est le cadre des travaux présentés dans [146]. Dans l'implantation réalisée, les communications avec opérateurs non constructeurs sont traitées au moyen de *gardes* à évaluer dynamiquement, suivant l'extension proposée en [159, 160]. Par la suite, cette extension a été rejetée en [165] du fait de l'adoption d'une nouvelle sémantique plus claire et plus homogène.

Si nous adoptons brièvement une perspective opérationnelle, nous remarquons qu'une communication ne peut avoir lieu que si les *deux* connecteurs sont prêts à échanger leurs valeurs ; il s'agit bien d'un mécanisme de *rendez-vous*. L'unificateur des messages donne la direction d'une communication, ou plus exactement, la direction de chaque échange à l'intérieur d'une communication. Soient en effet les messages suivants, de sorte $Seq[Nat]$:

$$A(0 <\!+\ s), \qquad B(x <\!+\ nil)$$

[23] Meije [2] et SCCS (Synchronous Calculus of Communicating Systems) [141] sont des formalismes inspirés de CCS, mais ce sont des calculs synchrones, alors que CCS est un calcul asynchrone.

[24] Rappelons que les équations entre constructeurs ne sont pas admises en FP2.

Le plus petit unificateur est $[s \mapsto nil, x \mapsto 0]$. Ceci montre qu'une communication peut être *bidirectionnelle*. Cette particularité de la communication de FP2 explique qu'il n'y ait pas de distinction entre connecteurs d'entrée et connecteurs de sortie — [159] montre pourquoi cette distinction, présente dans une précédente définition de FP2 [110], mène à une incohérence dans le traitement de l'opérateur de connexion[25]. Cette symétrie du rendez-vous se retrouve en LOTOS[26].

La connexion se généralise en une opération N-aire, qui permet de modéliser le rendez-vous à N par unification des N messages. Un cas particulier : la connexion d'un seul connecteur, qui consiste à le "reboucler" sur lui-même.

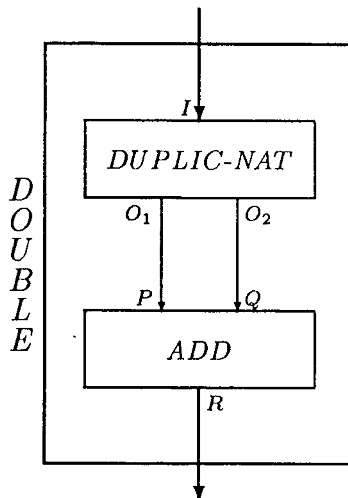
0.2.2.3 Abstraction

Soient P une présentation de processus et A un de ses connecteurs, le processus représenté par $P - A$ se comporte comme le processus représenté par P, mais toute communication extérieure au moyen de A est devenue impossible. Néanmoins, les précédentes connexions de A restent valides, puisque toutes les règles qui en ont résulté ne font plus mention de A. Il faut et il suffit donc, pour exprimer l'abstraction, d'enlever les règles où A apparaît.

$$tr\text{-}rules(P - A) = \{r \in tr\text{-}rules(P) \mid (\nexists u : A(u) \in event(r))\}$$

Remarque La connexion d'un seul connecteur et l'abstraction de ce même connecteur sont deux opérations différentes : elles considèrent toutes deux les règles dans lesquelles le connecteur apparaît, mais la première consiste à dupliquer ces règles en supprimant la référence à une communication sur ce connecteur, tandis que la seconde enlève ces règles purement et simplement.

Exemple 0.95 (Réseau de deux processus) *À l'aide d'un processus DUPLIC-NAT qui fabrique deux exemplaires d'un entier naturel et du processus ADD de l'exemple 0.92, nous allons construire un processus qui multiplie par 2 un entier naturel.*




```
process DUPLIC-NAT
   connectors  I, O1, O2  :  Nat
   states      A          :  –
   vars        n          :  Nat
   rules
      <>                           ==>  A
      <>  A  :  I(n) O1(n) O2(n)   ==>  A
endprocess

process DOUBLE is
   DUPLIC-NAT || ADD + O1.P + O2.Q – O1 – O2 – P – Q
endprocess
```

[25] Par exemple, pour un processus P et A, B, C, D, quatre de ses connecteurs, les expressions $P + A.B + C.D$ et $P + C.D + A.B$ ne désigneraient pas nécessairement le même processus.

[26] Par contre, il existe en LOTOS une distinction entre émission et réception. La règle est que si plusieurs émissions sont confrontées, le rendez-vous est possible si et seulement si toutes les valeurs émises sont identiques, et si plusieurs réceptions sont confrontées, la valeur commune qui circule est choisie de façon non déterministe.

Par application des formules précédentes, nous obtenons les règles suivantes pour la présentation du processus $DOUBLE$:

$$\begin{array}{rcrcl} & & & ==> & A.INIT \\ A.INIT & : & I(n) & ==> & A.A(n,n) \\ A.A(0,j) & : & R(j) & ==> & A.INIT \\ A.A(succ(i),j) & : & & ==> & A.A(i,succ(j)) \end{array}$$

Note Bien que la communication soit symétrique en FP2, nous clarifions les figures des exemples de processus en représentant une connexion par une flèche qui correspond au sens "intuitif".

0.2.2.4 Copie indicée

Elle permet la génération d'une famille de processus indicés, l'indice étant reporté sur les connecteurs. Ainsi ADD_1 possède les connecteurs P_1, Q_1, R_1, ADD_2 les connecteurs P_2, Q_2, R_2, ... Par ce biais, on peut construire un réseau comportant plusieurs exemplaires d'une même présentation de processus, sans risque de confusion entre les divers connecteurs.

0.2.2.5 Autres opérateurs

Les autres opérateurs proviennent d'autres langages et ont été ajoutés en vue d'applications particulières (calculs synchrones, gestion d'horloges). Nous les mentionnons rapidement par souci d'exhaustivité — tous ne sont pas implantés actuellement.

Notations utilisées : P, P_1, P_2 sont des présentations de processus, ev un événement, et n un entier naturel.

0.2.2.5.1 Composition asynchrone Une transition de $P_1 \mid P_2$ est soit une transition de P_1 seulement, soit une transition de P_2 seulement.

0.2.2.5.2 Composition synchrone Une transition de $P_1 \,|||\, P_2$ est une transition de P_1 et une transition de P_2 simultanées. Il existe en SCCS [141] (produit synchrone) et en LOTOS (opérateur "||").

0.2.2.5.3 Choix incontrôlable Cet opérateur n'est défini que pour deux présentations de processus P_1 et P_2 dont les signatures sont disjointes. Les règles initiales de $P_1 \;?\; P_2$ sont alors l'union disjointe des règles initiales de P_1 et de celles de P_2, et les règles de transition l'union disjointe des règles de transition de P_1 et de celles de P_2. En conséquence, c'est à l'initialisation que $P_1 \;?\; P_2$ "choisit", de façon non déterministe, s'il va se comporter comme P_1 ou comme P_2. Un opérateur de choix non déterministe entre deux comportements existe en CCS et SCCS (noté "+"), ainsi qu'en LOTOS ("[ ]").

0.2.2.5.4 Choix contrôlable Les règles initiales de $P_1 \;!\; P_2$ permettent que sa première transition puisse être soit une transition de P_1, soit une transition de P_2. Si cette première transition est une transition de P_1 (resp. P_2), par la suite $P_1 \;!\; P_2$ se comportera toujours comme P_1 (resp. P_2). L'appellation "choix contrôlable" est justifiée par le fait qu'il est possible d'influencer le choix si la première transition comporte un événement avec communications externes.

0.2.2.5.5 Déclenchement Les règles initiales de $ev \rightarrow P$ sont telles que sa première transition ne peut s'effectuer qu'à l'occurrence de l'événement *ev*. Ensuite, ce processus se comporte comme *P*.

0.2.2.5.6 Contrôle par événement Après l'initialisation, $ev \Rightarrow P$ se comporte comme *P*, mais chacune de ses transitions (y compris la première) doit s'effectuer conjointement aux occurrences de l'événement *ev*. C'est l'opérateur de *pilotage* de Meije.

0.2.2.5.7 *Time-out* L'état initial de P_1 % *n* % P_2 est choisi parmi les états initiaux de P_1 et le processus P_1 % *n* % P_2 se comporte comme P_1 pendant *n* transitions. Ensuite, s'il n'est pas terminé, la $(n+1)^{\text{ème}}$ transition le fait passer dans un état initial de P_2 et il se comporte ensuite comme P_2.

0.2.3 Processus et généricité

Si nous reprenons la présentation du processus *DUPLIC-NAT* de l'exemple 0.95, il est facile de voir que la sorte des connecteurs I, O_1, O_2 joue un rôle muet, comparable à la sorte des éléments d'une séquence lorsqu'on spécifie un opérateur tel que la concaténation (cf. exemple 0.87). En généralisant cette remarque, il semble dès lors souhaitable de pouvoir paramétrer une présentation de processus exactement comme on peut paramétrer un présentation de types abstraits algébriques, d'où l'idée de *processus génériques.*

Exemple 0.96 *Processus de duplication le plus général, du point de vue de la généricité.*

```
process DUPLIC req Ftype[t / ]
   connectors  I,O1,O2  : t
   states      A        : -
   vars        x        : t
   rules
      <>                            ==>  A
      <>  A  :  I(x) O1(x) O2(x)    ==>  A
endprocess
```

Exemple 0.97 *Processus, admettant un opérateur formel, et qui réalise une loi de composition interne sur deux entrées de même type.*

```
prop Groupoid[t / f]
   opns  f  :  t x t  ->  t
endprop
```

```
process DYADIC req Groupoid[t / f]
   connectors  P,Q,R  : t
   states      INIT   : -
   vars        x,y    : t
   rules
      <>                                  ==>  INIT
      <>  INIT  :  P(x) Q(y) R(f(x,y))    ==>  INIT
endprocess
```

Nous pouvons par conséquent donner une nouvelle définition de *DOUBLE* :

```
process DOUBLE is
  DUPLIC[Nat] || DYADIC.Groupoid[Nat / +]  + O1.P + O2.Q - O1 - O2 - P - Q
endprocess
```

et définir, de même :

process $SQUARE$ **is**
$DUPLIC[Nat] \parallel DYADIC.Groupoid[Nat / *] + O_1.P + O_2.Q - O_1 - O_2 - P - Q$
endprocess

En fait, ces "passages à la puissance" peuvent être à leur tour considérés comme des cas particuliers d'un processus générique :

process $POWER$ **req** $Groupoid[t / f]$ **is**
$DUPLIC[t] \parallel DYADIC.Groupoid[t / f] + O_1.P + O_2.Q - O_1 - O_2 - P - Q$
endprocess

Alors :

process $DOUBLE$ **is** $POWER.Groupoid[Nat / +]$
endprocess

process $SQUARE$ **is** $POWER.Groupoid[Nat / *]$
endprocess

Comme on l'aura remarqué, la notion de généricité est liée au départ à la notion de *réutilisabilité*. Des applications plus ambitieuses peuvent cependant être dégagées. En effet, de même que la généricité peut donner un cadre formel satisfaisant pour étudier des transformations de programmes fonctionnels [9], nous verrons au chapitre 5, que nous nous en servirons à propos des transformations de fonctions récursives en réseaux de processus parallèles. Nous pensons également que la généricité est une voie prometteuse pour bon nombre d'études, tant sur les fonctions que sur les processus, en permettant d'exprimer sur les signatures les hypothèses les plus faibles dont on a besoin. Étant donné que nous ne ferons pas, dans le cadre de cette thèse, une étude précise de la sémantique des processus génériques, nous considérerons que notre utilisation de cette possibilité, au chapitre 5, est uniquement *syntaxique*.

Partie I

Généricité

Chapitre 1

Les bases de la généricité

Christopher Strachey a introduit dans [171] le terme de **polymorphisme**, caractérisant l'emploi d'un même symbole pour désigner des fonctions différentes. Il distingue deux formes de polymorphisme :

- le polymorphisme *ad hoc* : au même symbole sont attachés plusieurs profils et à chaque profil correspond une définition différente : par exemple, c'est le cas si l'on note "+" à la fois l'addition entière et le "ou" booléen.

- le polymorphisme *paramétrique*, caractérisé par l'emploi de sortes et d'opérateurs variables. À ce titre, l'opérateur de composition, appliqué à deux opérateurs, est lui-même un opérateur polymorphe : pour toutes sortes s_1, s_2, s_3, et tous opérateurs $f : s_1 \rightarrow s_2$ et $g : s_2 \rightarrow s_3$, la composition de f et de g, $g \circ f$, admet le profil $s_1 \rightarrow s_3$. Par suite, le profil de "$\circ$" est :

$$((s_2 \rightarrow s_3) \times (s_1 \rightarrow s_2)) \rightarrow (s_1 \rightarrow s_3)$$

  pour toutes sortes s_1, s_2, s_3.

Note Dans l'introduction de [85], les définitions multiples par sortes ordonnées (cf. §0.1.2) sont présentées comme une forme supplémentaire de polymorphisme, qui procède par définitions *héritées* (au sens de l'inclusion des graphes).

La possibilité de surcharge dote FP2 du polymorphisme *ad hoc*. Au niveau sémantique, cela signifie que chaque couple opérateur/profil possède une interprétation différente. En fait, l'ensemble des opérateurs notés par un même symbole n'est jamais considéré globalement[27] : toute mention d'un opérateur, à l'intérieur d'une déclaration ou d'un terme, ne représente qu'un *seul* opérateur, à déterminer lors de l'analyse sémantique. C'est pourquoi nous comprenons la surcharge de FP2 comme une facilité purement syntaxique et n'en ferons plus mention dans le présent chapitre (nous l'étudions au chapitre 3 et notre choix y sera discuté).

---

[27] C'était par contre le cas dans une précédente définition de FP2 [114]. Une déclaration :

op g **is** f

avait pour effet de construire, pour chaque opérateur f, un opérateur g équivalent. Ce mécanisme, auquel s'ajoutaient des formes fonctionnelles dérivées du langage FP [4], a été abandonné en raison de sa complexité et de sa mauvaise coexistence avec la généricité.

Dans le domaine des spécifications algébriques, l'utilisation de présentations *génériques*[28], c'est-à-dire la paramétrisation d'une présentation par une présentation (dans laquelle peuvent figurer des sortes et des opérateurs), inclut une grande part de la puissance d'expression du polymorphisme paramétrique. Un avantage supplémentaire de ce mécanisme : il permet d'éviter la construction de termes d'ordre supérieur, dont la sémantique est difficile, alors que les opérateurs génériques suppléent dans de nombreux cas les opérateurs d'ordre supérieur. Cette particularité de la généricité est traitée par Joseph Goguen dans [73], et elle a été déjà utilisée dans le cadre de LPG [9]. Nous en rencontrerons beaucoup d'exemples.

Certains langages de programmation algorithmiques (par exemple, ADA [182]) ont intégré la généricité d'une manière uniquement syntaxique. La paramétrisation d'une procédure par des sortes et des opérateurs formels a alors pour principal but d'améliorer la *réutilisation* des programmes développés, au sens "réutilisation des textes sources". Dans le cadre des spécifications algébriques, un point important des présentations paramétrées est que leur utilisation n'est pas seulement syntaxique : elle répond aussi à des considérations sémantiques. Par exemple, il est possible de définir un type "séquence ordonnée d'éléments de type t", dont l'utilisation ne sera licite que pour un type t muni d'un ordre total. Poursuivant cette démarche, si nous définissons un opérateur d'ordre total sur les séquences ordonnées, la seule mention d'un ordre total pour un type donné t_0 permettra non seulement l'utilisation de séquences ordonnées d'éléments de type t_0, mais dotera l'algèbre de ce dernier type d'une relation d'ordre total. On voit dès lors que la généricité permet une plus grande abstraction et une formulation rigoureuse des hypothèses souhaitées sur les paramètres.

Nous commençons l'étude de la généricité par des rappels de la théorie développée dans [56, 172, 57, 58, 59], nous poursuivons par l'application à FP2, enfin, nous terminons ce chapitre par une comparaison avec les choix d'OBJ3.

1.1 Théorie de la généricité

Intuitivement, une présentation paramétrée se compose donc de deux descriptions : une description de la présentation *formelle* (qui contient tous les paramètres formels), et une description de la présentation *cible*. Ainsi, la définition du type "arbre binaire ordonné d'éléments de type t" doit comprendre d'une part la description la plus générale du type t et de sa relation d'ordre, d'autre part la spécification proprement dite du type des arbres binaires ordonnés. L'**instanciation** d'un tel type, c'est-à-dire le *remplacement* des paramètres formels par des paramètres effectifs, ne pourra s'effectuer qu'après *validation* de ce remplacement.

L'utilisation de telles présentations soulève plusieurs questions :

- quelle sémantique donner à une présentation paramétrée ?
- un type instancié représente-t-il la même algèbre que s'il avait été spécifié "directement", c'est-à-dire sans utilisation de la généricité ?

Afin de donner un sens à ces notions de présentation en paramètre et d'instanciation, nous allons introduire des outils sur les signatures et les présentations, ces outils se plaçant dans un contexte catégorique.

[28]Dans sa thèse [108], Paul Jacquet appelle *généricité incrémentale* un mécanisme de définition par surcharges successives et *généricité structurale* la donnée d'une définition unique dans laquelle figurent des paramètres sortes et opérateurs. Suivant la tendance actuelle, nous désignons par généricité uniquement la généricité structurale.

1.1.1 Morphismes de signatures — morphismes de présentations

Définition 1.1 *Soient* $\Sigma = (S, \Omega)$ *et* $\Sigma' = (S', \Omega')$ *deux signatures. Un* **morphisme de signatures** $h : \Sigma \to \Sigma'$ *est la donnée d'une fonction* $h_S : S \to S'$ *et d'une famille* h_Ω *de fonctions indicées par des éléments de* $S^* \times S$*, telles que chaque* $h_{s_1 \times \cdots \times s_n \to s}$ *ait pour profil* $\Omega_{s_1 \times \cdots \times s_n \to s} \to \Omega'_{h_S(s_1) \times \cdots \times h_S(s_n) \to h_S(s)}$.

Exemple 1.2 *Soient les signatures* Σ_{Mod_3} *(entiers modulo 3) et* Σ_{Bool}*, et soit* h' *défini comme suit :*

$$\Sigma_{Mod_3} \xrightarrow{h'} \Sigma_{Bool}$$

$$Mod_3 \longmapsto Bool$$

$$\begin{array}{rcrcl c rcrcl}
0 & : & - & \to & Mod_3 & \longmapsto & false & : & - & \to & Bool \\
1 & : & - & \to & Mod_3 & \longmapsto & true & : & - & \to & Bool \\
2 & : & - & \to & Mod_3 & \longmapsto & true & : & - & \to & Bool \\
min & : & Mod_3 \times Mod_3 & \to & Mod_3 & \longmapsto & and & : & Bool \times Bool & \to & Bool \\
max & : & Mod_3 \times Mod_3 & \to & Mod_3 & \longmapsto & or & : & Bool \times Bool & \to & Bool
\end{array}$$

h' est un morphisme de signatures.

Proposition 1.3 *Les signatures et les morphismes de signatures forment les objets et les flèches d'une catégorie notée* **Sig**.

Nous allons maintenant montrer comment étendre aux termes un morphisme de signatures h. $h : \Sigma \to \Sigma'$ induit un foncteur d'oubli $U_h : \mathbf{ALG}_{\Sigma'} \to \mathbf{ALG}_\Sigma$.

$$U_h(A') = A \qquad \text{est défini par} \quad \begin{cases} (\forall s \in S), & A_s = A'_{h_S(s)} \\ (\forall \omega \in \Omega_{__ \to s}), & A_\omega = A'_{h_{__ \to s}(\omega)} \\ (\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}), & \\ & A_\omega = A'_{h_{s_1 \times \cdots \times s_n \to s}(\omega)} \end{cases}$$

$$U_h(f' : A' \to B') = (f : A \to B) \quad \ldots \quad (\forall s \in S), f_s = f'_{h_S(s)}$$
$$\text{avec } A = U_h(A'), B = U_h(B')$$

pour $A', B' \in \text{Ob}\ \mathbf{ALG}_{\Sigma'}, f' \in \text{Arr}\ \mathbf{ALG}_{\Sigma'}$.

Soit V un ensemble de variables indicées par des éléments de S, on construit un ensemble $V^\sharp$ de variables indicées par des éléments de S' par :

$$V^\sharp_{s'} = \bigcup_{h_S(s) = s'} V_s$$

Notons $h^\sharp : T_\Sigma(V) \to U_h(T_{\Sigma'}(V^\sharp))$ l'extension de l'injection canonique $V \hookrightarrow U_h(T_{\Sigma'}(V^\sharp))$ par la propriété d'algèbre libre de $T_\Sigma(V)$. L'extension $h^\sharp$ vérifie donc :

$$h^\sharp(\overline{\omega}) = \overline{h(\omega)}$$
$$h^\sharp(\overline{\omega(t_1, \ldots, t_n)}) = \overline{h(\omega)(h^\sharp(t_1), \ldots, h^\sharp(t_n))}$$

Définition 1.4 *Soient $\mathcal{P} = (S, \Omega, E)$ et $\mathcal{P}' = (S', \Omega', E')$ deux présentations, et soit $h : (S, \Omega) \rightarrow (S', \Omega')$ un morphisme de signatures. h est un* **morphisme de présentations** *si et seulement si h vérifie la propriété :*

$$(\forall (t_1 == t_2) \in E), (h^\sharp(t_1) == h^\sharp(t_2)) \textit{ est une } \text{conséquence logique} \textit{ des équations de } E'$$

Remarque Du point de vue de la validation, la notion de *conséquence logique* suppose l'existence de *règles de déduction*. Les problèmes liés à l'établissement de telles règles ne seront pas abordés ici. Nous admettrons simplement que nous disposons d'un système de règles **adéquat**, c'est-à-dire tel que toute équation est déductible d'un ensemble d'équations F si et seulement si elle est vraie dans tous les modèles de F :

$$(F \vdash (t_1 == t_2)) \Longleftrightarrow (F \models (t_1 == t_2))$$

"$\vdash$" désignant la relation de déduction.

Notons que notre définition est, d'un certain point de vue, assez proche de celle de [59] : pour toute équation $(t_1 == t_2)$ de E, $(h^\sharp(t_1) == h^\sharp(t_2))$ se déduit des équations de E' par raisonnement équationnel[29]. Ces deux définitions précédentes sont plus générales que celle de [58] : $(\forall (t_1 == t_2) \in E), (h^\sharp(t_1) == h^\sharp(t_2)) \in E'$.

Exemple 1.5 (Suite de l'exemple 1.2) *Reprenons les signatures $\Sigma_{Mod_3}, \Sigma_{Bool}$, et le morphisme de signatures $h' : \Sigma_{Mod_3} \rightarrow \Sigma_{Bool}$. Donnons, pour Σ_{Mod_3}, des équations qui définissent min et max en fonction de $0, 1, 2$, et, pour Σ_{Bool}, des équations qui définissent and et or en fonction de true et false :*

$$\begin{aligned} 0 \; min \; i &== 0 \\ 1 \; min \; 0 &== 0 \\ &\vdots \\ 0 \; max \; i &== i \\ &\vdots \end{aligned} \qquad \begin{aligned} false \; and \; b &== false \\ true \; and \; b &== b \\ false \; or \; b &== b \\ true \; or \; b &== b \end{aligned}$$

pour $i : Mod_3$ et $b : Bool$. Il est assez facile de voir que h' est alors un morphisme de présentations.

Proposition 1.6 *Les présentations et les morphismes de présentations forment les objets et les flèches d'une catégorie notée* **Pres**.

1.1.2 Présentations paramétrées

Avant de rappeler les définitions, nous allons expliciter les notations abrégées utilisées. Soient $\mathcal{P}_0 = (S_0, \Omega_0, E_0)$ et $\mathcal{P} = (S_0 \uplus S, \Omega_0 \uplus \Omega, E_0 \uplus E)$, deux présentations. Nous noterons globalement les unions disjointes :

$$\mathcal{P} = \mathcal{P}_0 \uplus (S, \Omega, E)$$

Intuitivement, un type abstrait paramétré est une construction qui fait correspondre à chaque algèbre satisfaisant la présentation paramètre une algèbre satisfaisant la présentation cible.

Définition 1.7 *Une* **présentation paramétrée** *$(\mathcal{P}_0, \mathcal{P})$ est la donnée :*

- *d'une présentation paramètre $\mathcal{P}_0 = (S_0, \Omega_0, E_0)$,*
- *d'une présentation cible $\mathcal{P} = \mathcal{P}_0 \uplus (S, \Omega, E)$, que nous noterons également $\mathcal{P}(\mathcal{P}_0)$.*

[29] Le raisonnement équationnel, ou remplacement d'égaux par égaux, est un système de déduction adéquat [25].

La sémantique d'une telle présentation est donné par un foncteur libre $\mathbf{ALG}_{\mathcal{P}_0} \to \mathbf{ALG}_{\mathcal{P}}$*, adjoint à gauche au foncteur d'oubli* $\mathbf{ALG}_{\mathcal{P}} \to \mathbf{ALG}_{\mathcal{P}_0}$*. (Rappelons qu'un tel foncteur libre est unique à un isomorphisme naturel près.)*

La preuve d'existence et la construction d'un foncteur libre $F : \mathbf{ALG}_{\mathcal{P}_0} \to \mathbf{ALG}_{\mathcal{P}}$ engendré par le foncteur d'oubli $U : \mathbf{ALG}_{\mathcal{P}} \to \mathbf{ALG}_{\mathcal{P}_0}$ est donnée en [172, 59]. En fait, cette construction, qui généralise la notion d'algèbre de termes satisfaisant un ensemble d'équations, est analogue à celle du foncteur de synthèse qui relie les modèles d'une présentation aux modèles d'une extension de cette présentation. C'est-à-dire que pour toute $\mathcal{P}_0$-algèbre A_0 :

$$F(A_0) = \frac{T_{(S_0 \uplus S, \Omega_0 \uplus \Omega)}(U_{S_0}(A_0))}{\equiv_{E_0 \uplus E, \sim_{A_0}}} \tag{1.1}$$

Nous renvoyons au §A.2.5 pour la description complète de cette construction et des notations que nous avons utilisées. De même, l'image d'un (S_0, Ω_0)-homomorphisme s'explicite de façon tout à fait identique : cf. §A.2.5 pour plus de détails.

Définition 1.8 *Soit* $(\mathcal{P}'_0, \mathcal{P}')$ *une présentation paramétrée, et soit un foncteur* $F' : \mathbf{ALG}_{\mathcal{P}'_0} \to \mathbf{ALG}_{\mathcal{P}'}$*. Soit* $(\mathcal{P}_0, \mathcal{P})$ *une présentation paramétrée telle que* $\mathcal{P}'_0 \subseteq \mathcal{P}_0$ *et* $\mathcal{P}' \subseteq \mathcal{P}$*.* $(\mathcal{P}_0, \mathcal{P})$ *est* **correcte par rapport à** F' *si et seulement si pour tout foncteur libre* $F : \mathbf{ALG}_{\mathcal{P}_0} \to \mathbf{ALG}_{\mathcal{P}}$*, adjoint à gauche au foncteur d'oubli* $U : \mathbf{ALG}_{\mathcal{P}} \to \mathbf{ALG}_{\mathcal{P}_0}$*, le diagramme suivant est commutatif, à un isomorphisme naturel près :*

$$\begin{array}{ccc} \mathbf{ALG}_{\mathcal{P}'_0} & \xrightarrow{F'} & \mathbf{ALG}_{\mathcal{P}'} \\ V_0 \uparrow & \cong & \uparrow V \\ \mathbf{ALG}_{\mathcal{P}_0} & \xrightarrow{F} & \mathbf{ALG}_{\mathcal{P}} \end{array}$$

c'est-à-dire :

$$F' \circ V_0 \cong V \circ F$$

V_0 *(resp.* V*) étant le foncteur d'oubli* $\mathbf{ALG}_{\mathcal{P}_0} \to \mathbf{ALG}_{\mathcal{P}'_0}$ *(resp.* $\mathbf{ALG}_{\mathcal{P}} \to \mathbf{ALG}_{\mathcal{P}'}$*).*

Si le diagramme est commutatif, alors $(\mathcal{P}_0, \mathcal{P})$ *est dite* **fortement correcte par rapport à** F'.

La donnée d'un foncteur comme sémantique d'une présentation paramétrée concrétise effectivement l'idée intuitive que nous en avions : à un modèle de la présentation paramètre, on fait correspondre un modèle de la présentation cible. Dès lors que cette sémantique consiste en la donnée, à un isomorphisme naturel près, d'un foncteur, il en découle que la correction d'une spécification paramétrée s'exprime par rapport à un foncteur reliant deux catégories d'algèbres, et c'est exactement ce que traduit la définition précédente. Remarquons qu'elle inclut l'occurrence d'opérateurs cachés dans la présentation paramétrée $(\mathcal{P}_0, \mathcal{P})$. Il est également important de noter que ces deux définitions — la sémantique d'une présentation paramétrée et la correction par rapport à cette sémantique — généralisent le cas non paramétré. On peut dans ce cas considérer qu'une présentation $\mathcal{P}$ est paramétrée par la présentation $(\emptyset, \emptyset, \emptyset)$ et que le foncteur libre est alors :

$$F : \mathbf{1} \to \mathbf{ALG}_{\mathcal{P}}$$

où "**1**" est la catégorie terminale qui admet $\emptyset$ pour seul objet et $id_{\emptyset}$ pour seule flèche. L'image d'un objet initial par un foncteur libre étant un objet initial, on obtient que l'image de l'objet $\emptyset$ est une

algèbre initiale de $\mathbf{ALG}_{\mathcal{P}}$. En fait, si l'on applique la formule (1.1), on retrouve bien $F(\emptyset) = T_{\mathcal{P}}$. La correction par rapport à F d'une présentation $((\emptyset, \emptyset, \emptyset), \mathcal{P}_1)$ équivaut à la coïncidence des algèbres initiales de $\mathbf{ALG}_{\mathcal{P}_1}$ et de $\mathbf{ALG}_{\mathcal{P}}$. Remarquons enfin qu'il n'est pas judicieux de donner une sémantique initiale à une présentation paramètre dès lors qu'elle n'offre pas nécessairement le moyen de *générer* des éléments à l'intérieur d'une algèbre. Ainsi, le modèle initial de la présentation *Ftype* (cf. exemple 0.86) est l'algèbre $(\emptyset, \emptyset)$. C'est pourquoi on donne à une présentation paramètre une sémantique lâche (*loose semantics*), considérant *tous* ses modèles.

Jusqu'à présent, nous n'avons pas posé d'hypothèse exprimant que la présentation paramètre est protégée dans la présentation cible. Donnons les définitions.

Définition 1.9 *Soit* $(\mathcal{P}_0, \mathcal{P})$ *une présentation paramétrée, et soit* F' *un foncteur* $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$. F' *est dit :*

- **persistant** *s'il existe un isomorphisme naturel* $U \circ F' \cong id_{\mathbf{ALG}_{\mathcal{P}_0}}$,
- **fortement persistant** *si* $U \circ F' = id_{\mathbf{ALG}_{\mathcal{P}_0}}$,

U *étant le foncteur d'oubli* $\mathbf{ALG}_{\mathcal{P}} \rightarrow \mathbf{ALG}_{\mathcal{P}_0}$.

$(\mathcal{P}_0, \mathcal{P})$ *est dite persistante (resp. fortement persistante) si un foncteur libre quelconque* F : $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$, *sémantique de* $(\mathcal{P}_0, \mathcal{P})$, *est persistant (resp. fortement persistant).*

Remarque Deux foncteurs libres quelconques $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$ étant identiques à un isomorphisme naturel près, il s'ensuit qu'un foncteur libre $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$ est persistant si et seulement si tout foncteur libre $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$ est persistant.

Exemple 1.10 *Présentation paramétrée avec foncteur libre non persistant. Soit la présentation paramètre suivante :*

presentation $\mathcal{PN}_0$ **is**

sorts	$= \{Bool, Nat\}$					
opns	$true, false$	:		$-$	$\rightarrow$	$Bool$
	not	:	$Bool$		$\rightarrow$	$Bool$
	and, or	:	$Bool$	$\times$ $Bool$	$\rightarrow$	$Bool$
	0	:		$-$	$\rightarrow$	Nat
	$succ$	:	Nat		$\rightarrow$	Nat
	$\leqslant$	:	Nat	$\times$ Nat	$\rightarrow$	$Bool$
vars	i, j	:	Nat			

eqns

$\langle\langle$*équations définissant* $not, and, or\rangle\rangle$

<>	0	$\leqslant$	0	$==$	$true$
<>	0	$\leqslant$	$succ(j)$	$==$	$true$
<>	$succ(i)$	$\leqslant$	0	$==$	$false$
<>	$succ(i)$	$\leqslant$	$succ(j)$	$==$	$i \leqslant j$

end $\mathcal{PN}_0$

et soit la présentation cible :

presentation $\mathcal{PN}$ **is** $\mathcal{PN}_0 \uplus$

opns $\top$: $-$ $\rightarrow$ *Nat*

vars i : *Nat*

eqns

$<>$ $i \leqslant \top$ $==$ *true*

$<>$ $\top \leqslant 0$ $==$ *false*

$<>$ $\top \leqslant succ(i)$ $==$ *false*

$<>$ $\top \leqslant \top$ $==$ *true*

end $\mathcal{PN}$

Soit :

$$\mathbf{ALG}_{\mathcal{PN}_0} \underset{U}{\overset{F}{\rightleftarrows}} \mathbf{ALG}_{\mathcal{PN}}$$

où F est le foncteur libre construit d'après la formule (1.1). Alors $(U \circ F(T_{\mathcal{PN}_0}))_{Nat}$ est formé des termes :

$$0 \leqslant succ(0) \leqslant succ^2(0) \leqslant \ldots \leqslant \top$$

C'est-à-dire que l'on a ajouté un élément maximum au type *Nat*. $(T_{\mathcal{PN}_0})_{Nat}$ n'admettant pas d'élément maximum pour "$\leqslant$", il ne saurait exister d'homomorphisme :

$$g : U \circ F(T_{\mathcal{PN}_0}) \rightarrow T_{\mathcal{PN}_0}$$

tel que :

$$g_{Nat}(i \leqslant \top) = g_{Nat}(i) \leqslant g_{Nat}(\top)$$

pour tout $i \in (T_{\mathcal{PN}_0})_{Nat}$. Par suite, il n'existe pas d'isomorphisme naturel de $U \circ F$ vers $id_{\mathbf{ALG}_{\mathcal{PN}_0}}$ et donc, le foncteur F n'est pas persistant.

Remarque En fait, l'exemple précédent peut être compris comme une importation de présentation, en considérant une sémantique lâche pour la présentation $\mathcal{PN}_0$. Nous l'avons présenté sous cette forme pour des raisons de simplicité, mais il peut se généraliser en remplaçant la sorte *Nat* par une sorte t caractérisant un type totalement ordonné (cf. exemple 1.38).

Le résultat suivant permet de ne considérer que des foncteurs fortement persistants.

Lemme 1.11 ([59]) *Soit* $(\mathcal{P}_0, \mathcal{P})$ *une présentation paramétrée, et soit un foncteur persistant* F : $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$. *Il existe un foncteur* F' : $\mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}}$, *fortement persistant, et tel que* $F' \cong F$.

1.1.3 Passage de paramètre standard

Soit $(\mathcal{P}_0, \mathcal{P})$ une présentation paramétrée, $\mathcal{P}_0$ représente intuitivement la présentation "la plus générale" exigée pour construire $\mathcal{P}$. Nous allons maintenant nous intéresser à l'instanciation, c'est-à-dire au remplacement des paramètres formels par des paramètres effectifs, autrement dit à la substitution des sortes et des opérateurs de $\mathcal{P}_0$ par ceux d'une présentation effective $\mathcal{P}_1$. Suivant la démarche de [58, 59], nous donnons le cas *standard*, lorsque les paramètres effectifs constituent une présentation simple, non paramétrée. Dans le §1.1.4, nous traiterons le cas général, lorsque la présentation en paramètre effectif est elle-même une présentation paramétrée, et verrons alors que le premier cas

est inclus dans le second. Pour l'instant, un exemple simple du cas que nous allons aborder est l'instanciation de la présentation paramétrée (*Ring*, *Matrix*) — où *Ring* caractérise tout anneau unitaire commutatif et *Matrix* désigne la construction de matrices d'éléments de sorte *Ring* — par la présentation *Nat* dans laquelle la sorte *Nat* est munie des opérateurs "+" et "*".

Définition 1.12 (Syntaxe) *Soit* $(\mathcal{P}_0, \mathcal{P})$ *une présentation paramétrée, avec* $\mathcal{P} = \mathcal{P}_0 \uplus (S, \Omega, E)$*, et soit* $\mathcal{P}_1 = (S_1, \Omega_1, E_1)$ *une présentation telle qu'il existe un morphisme de présentations* $h : \mathcal{P}_0 \to \mathcal{P}_1$*. Le* **passage de paramètre standard** *s'effectue d'après le diagramme suivant dans la catégorie* **Pres** *(les morphismes de présentations* s_0 *et* s_1 *sont en fait des injections canoniques) :*

$$\begin{array}{ccc} \mathcal{P}_0 & \overset{s_0}{\hookrightarrow} & \mathcal{P}(\mathcal{P}_0) \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \hbar} \\ \mathcal{P}_1 & \underset{s_1}{\hookrightarrow} & \mathcal{P}(\mathcal{P}_1) \end{array}$$

avec :

$$\mathcal{P}(\mathcal{P}_1) = \mathcal{P}_1 \uplus (S, \hbar(\Omega), \hbar^\sharp(E))$$

et le morphisme de présentations $\hbar$ *est défini comme suit :*

$$\hbar_S(s) = \textbf{if } s \in S \textbf{ then } s \textbf{ else } h_S(s) \textbf{ endif} \tag{1.2}$$

$$\hbar_\Omega(\omega) = \textbf{if } \omega : s_1 \times \cdots \times s_n \to s \in \Omega \textbf{ then } \omega : \hbar_S(s_1) \times \cdots \times \hbar_S(s_n) \to \hbar_S(s) \textbf{ else } h_\Omega(\omega) \textbf{ endif} \tag{1.3}$$

Cette définition constitue un cas particulier d'une construction de la théorie des catégories.

Théorème 1.13 ([58, 59]) *Le diagramme précédent de passage de paramètre standard définit une somme amalgamée*[30] *:*

$$\mathcal{P}(\mathcal{P}_1) = \mathcal{P}(\mathcal{P}_0) \coprod_{s_0,h} \mathcal{P}_1$$

c'est-à-dire que pour toute présentation $\mathcal{P}'$ *et pour tous morphismes de présentations* $s' : \mathcal{P}_1 \to \mathcal{P}'$ *et* $h' : \mathcal{P}(\mathcal{P}_0) \to \mathcal{P}'$*, tels que* $s' \circ h = h' \circ s_0$*, il existe un unique morphisme de présentations* $f : \mathcal{P}(\mathcal{P}_1) \to \mathcal{P}'$*, tel que* $f \circ s_1 = s'$ *et* $f \circ \hbar = h'$*.*

Note D'un point de vue pratique, [152] souligne que cette construction de somme amalgamée *duplique* les sortes et les opérateurs présents à la fois dans $\mathcal{P}$ et dans $\mathcal{P}_1$, mais non dans $\mathcal{P}_0$. Ce problème ne sera pas traité ici.

Définition 1.14 *Soient* $(\mathcal{P}_0, \mathcal{P})$ *une présentation paramétrée et* F *un foncteur libre* $\mathbf{ALG}_{\mathcal{P}_0} \to \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}$*. Soit* $\mathcal{P}_1$ *une présentation telle qu'il existe un morphisme de présentations* $h : \mathcal{P}_0 \to \mathcal{P}_1$*.*

- *La* **sémantique** *du passage de paramètre standard est :*

$$(F, T_{\mathcal{P}_1}, T_{\mathcal{P}(\mathcal{P}_1)})$$

$T_{\mathcal{P}_1}$ *et* $T_{\mathcal{P}(\mathcal{P}_1)}$ *étant les algèbres initiales de* $\mathcal{P}_1$ *et de* $\mathcal{P}(\mathcal{P}_1)$*, à un isomorphisme près.*

[30] Cf. §A.1.3.1.

- *Notons U_{s_0}, U_h, U_{s_1}, $U_{\hbar}$ les foncteurs d'oubli induits respectivement par s_0, h, s_1, $\hbar$. Le passage de paramètre standard est dit* **correct** *si les conditions suivantes sont satisfaites :*
  - ⋆ *protection du paramètre :* $U_{s_1}(T_{\mathcal{P}(\mathcal{P}_1)}) = T_{\mathcal{P}_1}$,
  - ⋆ *compatibilité du passage de paramètre standard avec la sémantique de la présentation paramétrée :*

$$F \circ U_h(T_{\mathcal{P}_1}) = U_{\hbar}(T_{\mathcal{P}(\mathcal{P}_1)})$$

Nous allons maintenant aborder le niveau sémantique, et montrer une possibilité de construire, à partir d'une algèbre qui est un modèle d'une présentation paramètre, une algèbre modèle de la présentation cible dans laquelle a été effectuée un passage de paramètre standard.

Définition 1.15 ([59]) *Considérons le passage de paramètre standard défini par le diagramme suivant :*

$$\begin{array}{ccc} \mathcal{P}_0 & \stackrel{s_0}{\hookrightarrow} & \mathcal{P}(\mathcal{P}_0) \\ h \downarrow & & \downarrow \hbar \\ \mathcal{P}_1 & \stackrel{s_1}{\hookrightarrow} & \mathcal{P}(\mathcal{P}_1) \end{array}$$

- *Soit $A_0 \in \mathbf{ALG}_{\mathcal{P}_0}$, et soient $A \in \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}$ et $A_1 \in \mathbf{ALG}_{\mathcal{P}_1}$ telles que :*

$$U_{s_0}(A) = U_h(A_1) = A_0$$

La **somme amalgamée de** A **et de** A_1 **suivant** A_0, *notée :*

$$A' \stackrel{\text{déf}}{=} A \coprod_{A_0} A_1$$

est définie par :

$$\begin{array}{rcl} A'_s & = & \textbf{if } s \in S_1 \textbf{ then } (A_1)_s \textbf{ else } A_s \\ & & \textbf{endif} \\ A'_\omega & = & \textbf{if } \omega \in \Omega_1 \textbf{ then } (A_1)_\omega \textbf{ else } A_\omega \\ & & \textbf{endif} \end{array}$$

- *Soit $f_0 \in$ Arr $\mathbf{ALG}_{\mathcal{P}_0}$, et soient $f \in$ Arr $\mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}$ et $f_1 \in$ Arr $\mathbf{ALG}_{\mathcal{P}_1}$ telles que :*

$$U_{s_0}(f) = U_h(f_1) = f_0$$

La somme amalgamée de f et de f_1 suivant f_0 est définie par :

$$\begin{array}{rcl} (f \coprod_{f_0} f_1)_s & = & \textbf{if } s \in S_1 \textbf{ then } (f_1)_s \textbf{ else } f_s \\ & & \textbf{endif} \end{array}$$

Théorème 1.16 ([59]) *Soit $(\mathcal{P}_0, \mathcal{P})$ une présentation paramétrée persistante, et soit $\mathcal{P}_1$ une présentation telle qu'il existe un morphisme de présentations $h : \mathcal{P}_0 \to \mathcal{P}_1$.*

- *Le passage de paramètre standard est correct.*
- *Soit F le foncteur libre $\mathbf{ALG}_{\mathcal{P}_0} \to \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}$, sémantique de $(\mathcal{P}_0, \mathcal{P})$, et soit $A_0 = U_h(T_{\mathcal{P}_1})$.*

$$T_{\mathcal{P}_1} \coprod_{A_0} F(A_0) \cong T_{\mathcal{P}(\mathcal{P}_1)}$$

De plus, la notion de somme amalgamée d'algèbres s'étend aux catégories d'algèbres considérées. Soit le diagramme suivant dans $\mathbf{Cat}'$, la catégorie des grandes catégories[31] :

$$\begin{array}{ccc}
\mathbf{ALG}_{\mathcal{P}_0} & \underset{F}{\overset{U_{s_0}}{\rightleftarrows}} & \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)} \\
\uparrow U_h & & \uparrow U_h \\
\mathbf{ALG}_{\mathcal{P}_1} & \underset{U_{s_1}}{\overset{F'}{\rightleftarrows}} & \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_1)}
\end{array}$$

avec :

$$\mathbf{ALG}_{\mathcal{P}(\mathcal{P}_1)} \stackrel{\text{déf}}{=} \mathbf{ALG}_{\mathcal{P}_1} \coprod_{\mathbf{ALG}_{\mathcal{P}_0}} \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}$$

la catégorie dont les objets et les flèches sont :

$$A_1 \coprod_{A_0} A, \text{ avec } A_1 \in \text{Ob } \mathbf{ALG}_{\mathcal{P}_1}, \quad A \in \text{Ob } \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}, \quad U_{s_0}(A) = U_h(A_1) = A_0$$

$$f_1 \coprod_{f_0} f, \text{ avec } f_1 \in \text{Arr } \mathbf{ALG}_{\mathcal{P}_1}, \quad f \in \text{Arr } \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}, \quad U_{s_0}(f) = U_h(f_1) = f_0$$

F étant le foncteur libre sémantique de $(\mathcal{P}_0, \mathcal{P})$, le foncteur F' est l'**extension** de F par h, noté :

$$F' = \text{Ext}(F, h)$$

et est construit comme suit :

$$\begin{aligned}
F'(A_1) &= A_1 \coprod_{A_0} F(A_0) \\
F'(f_1) &= f_1 \coprod_{f_0} F(f_0)
\end{aligned}$$

[31] Cf. annexe A, note du §A.2.1. Nous postulons que les supports de tous les modèles d'une présentation appartiennent à un univers $\mathcal{U}$. Ainsi, les modèles d'une présentation constituent une grande catégorie. À noter qu'en général, ils ne forment pas une petite catégorie. Considérons, par exemple, la présentation $(\{s\}, \emptyset, \emptyset)$. Toute algèbre composée d'un petit ensemble de $\mathcal{U}$ est un modèle de cette présentation. Les objets de cette catégorie forment une classe, mais non un petit ensemble, car on aurait dans ce cas $\mathcal{U} \in \mathcal{U}$.

En outre, le diagramme formé des U_{s_0}, U_h, U_{s_1}, $U_{\hbar}$ définit un produit fibré :

$$\mathbf{ALG}_{\mathcal{P}(\mathcal{P}_1)} = \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)} \prod_{U_{s_0},U_h} \mathbf{ALG}_{\mathcal{P}_1}$$

et il a été démontré [59] que F' est un foncteur libre adjoint à gauche de U_{s_1}, tel que :

- $U_{\hbar} \circ F' = F \circ U_h$,
- si F est persistant (resp. fortement persistant), alors F' est persistant (resp. fortement persistant).

Remarquons que F' étant un foncteur libre, il s'ensuit que l'image d'une algèbre initiale de $\mathbf{ALG}_{\mathcal{P}_1}$ par F' est une algèbre initiale de $\mathbf{ALG}_{\mathcal{P}(\mathcal{P}_1)}$.

1.1.4 Passage de paramètre générique

Cette fois, nous passons en paramètre non pas une présentation simple, mais une présentation paramétrée. Un exemple est le passage du paramètre $(Ring, Matrix)$ pour la présentation paramétrée $(Param, Seq)$, en vue de construire les séquences dont les éléments sont des matrices d'éléments de sorte $Ring$.

Définition 1.17 (Syntaxe) *Soient deux présentations paramétrées* $(\mathcal{P}_0, \mathcal{P})$, *avec* $\mathcal{P} = \mathcal{P}_0 \uplus (S, \Omega, E)$, *et* $(\mathcal{P}_1, \mathcal{P}')$, *telles qu'il existe un morphisme de présentations* $h : \mathcal{P}_0 \to \mathcal{P}'$. *Le* **passage de paramètre générique** *s'effectue d'après le diagramme suivant dans la catégorie* **Pres** *:*

$$\begin{array}{ccccc}
 & & \mathcal{P}_0 & \xrightarrow{s_0} & \mathcal{P}(\mathcal{P}_0) \\
 & & \downarrow h & & \downarrow \hbar \\
\mathcal{P}_1 & \xrightarrow{s'} & \mathcal{P}'(\mathcal{P}_1) & \xrightarrow{s_1} & \mathcal{P} *_h \mathcal{P}'(\mathcal{P}_1)
\end{array}$$

avec :

$$\mathcal{P} *_h \mathcal{P}'(\mathcal{P}_1) = \mathcal{P}'(\mathcal{P}_1) \uplus (S, \hbar(\Omega), \hbar^{\sharp}(E))$$

$(\mathcal{P}_1, \mathcal{P} *_h \mathcal{P}')$ *est la présentation paramétrée obtenue : c'est la* **composition** *des deux présentations paramétrées* $(\mathcal{P}_0, \mathcal{P})$ *et* $(\mathcal{P}_1, \mathcal{P}')$.

La définition de $\hbar$ *est la même que pour le passage de paramètre standard — cf. les équations (1.2) et (1.3) :*

$$\begin{array}{rcl}
\hbar_S(s) & = & \textbf{if } s \in S \textbf{ then } s \textbf{ else } h_S(s) \\
 & & \textbf{endif} \\
\hbar_\Omega(\omega) & = & \textbf{if } \omega : s_1 \times \cdots \times s_n \to s \in \Omega \textbf{ then } \omega : \hbar_S(s_1) \times \cdots \times \hbar_S(s_n) \to \hbar_S(s) \textbf{ else } h_\Omega(\omega) \\
 & & \textbf{endif}
\end{array}$$

De même que le diagramme de passage de paramètre standard, ce diagramme définit une somme amalgamée :

$$\mathcal{P} *_h \mathcal{P}'(\mathcal{P}_1) = \mathcal{P}(\mathcal{P}_0) \coprod_{s_0,h} \mathcal{P}'(\mathcal{P}_1)$$

Définition 1.18 *Soient deux présentations paramétrées* $(\mathcal{P}_0, \mathcal{P})$ *et* $(\mathcal{P}_1, \mathcal{P}')$*, telles qu'il existe un morphisme de présentations* $h : \mathcal{P}_0 \to \mathcal{P}'$. *Soient également les foncteurs libres :*

$$\begin{array}{rcccll} F_0 & : & \mathbf{ALG}_{\mathcal{P}_0} & \to & \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)}, & \textit{sémantique de } (\mathcal{P}_0,\mathcal{P}) \\ F_1 & : & \mathbf{ALG}_{\mathcal{P}_1} & \to & \mathbf{ALG}_{\mathcal{P}'(\mathcal{P}_1)}, & \quad \ldots \quad (\mathcal{P}_1,\mathcal{P}') \\ F_0 *_h F_1 & : & \mathbf{ALG}_{\mathcal{P}_1} & \to & \mathbf{ALG}_{\mathcal{P}*_h\mathcal{P}'(\mathcal{P}_1)}, & \quad \ldots \quad (\mathcal{P}_1,\mathcal{P} *_h \mathcal{P}') \end{array}$$

Alors :

- *La* **sémantique** *du passage de paramètre générique est :*

$$(F_0, F_1, F_0 *_h F_1)$$

- *Le passage de paramètre générique est* **correct** *si les conditions suivantes sont satisfaites (remarquons qu'elles sont analogues à celles du passage de paramètre standard) :*
  - ⋆ *protection du paramètre :* $U_{s_1} \circ (F_0 *_h F_1) = F_1$,
  - ⋆ *compatibilité du passage de paramètre générique avec les sémantiques des présentations paramétrées* $(\mathcal{P}_0, \mathcal{P})$ *et* $(\mathcal{P}_1, \mathcal{P}')$ *:*

$$U_h \circ (F_0 *_h F_1) = F_0 \circ U_h \circ F_1$$

Théorème 1.19 ([59]) *Soient deux présentations paramétrées persistantes* $(\mathcal{P}_0, \mathcal{P})$ *et* $(\mathcal{P}_1, \mathcal{P}')$*, telles qu'il existe un morphisme de présentations* $h : \mathcal{P}_0 \to \mathcal{P}'$. *Soient également* F_0 *et* F_1 *les foncteurs libres qui représentent respectivement les sémantiques de* $(\mathcal{P}_0, \mathcal{P})$ *et de* $(\mathcal{P}_1, \mathcal{P}')$.

- *Le passage de paramètre générique est correct.*
- *La présentation paramétrée* $(\mathcal{P}_1, \mathcal{P} *_h \mathcal{P}')$ *est persistante.*
- $F_0 *_h F_1 = \mathrm{Ext}(F_0, h) \circ F_1$.

Remarque Si les foncteurs F_0 et F_1 sont fortement persistants, alors le foncteur $F_0 *_h F_1$ est lui aussi fortement persistant.

Si l'on pose $\mathcal{P}_1 = (\emptyset, \emptyset, \emptyset)$ dans la définition précédente, alors on retrouve exactement la définition et les propriétés du passage de paramètre standard. Il s'ensuit que du point de vue du passage de paramètre, toute présentation au sens classique peut être considérée comme une présentation paramétrée par la présentation $(\emptyset, \emptyset, \emptyset)$. Nous avions déjà remarqué un fait analogue au §1.1.2. Dans toute la suite, nous noterons une telle présentation paramètre par "◇".

1.1.5 Composition de paramétrisations

Nous allons à présent étendre les résultats dégagés précédemment à la composition de présentations paramétrées. Le point important est que le résultat d'une composition de passages de paramètres, qu'ils soient standards ou génériques, est indépendant de la construction choisie pour le calcul, la correction par rapport aux modèles sémantiques étant préservée.

Théorème 1.20 ([58, 59]) *La composition de présentations paramétrées persistantes est associative.*

Plus concrètement, étant donné trois présentations paramétrées persistantes $(\mathcal{P}_0, \mathcal{P})$, $(\mathcal{P}_1, \mathcal{P}')$, $(\mathcal{P}_2, \mathcal{P}'')$, et deux morphismes de présentations :

$$
\begin{array}{rcl}
h & : & \mathcal{P}_0 \rightarrow \mathcal{P}' \\
h' & : & \mathcal{P}_1 \rightarrow \mathcal{P}''
\end{array}
$$

Soient également les foncteurs libres :

$$
\begin{array}{rcl}
F_0 & : & \mathbf{ALG}_{\mathcal{P}_0} \rightarrow \mathbf{ALG}_{\mathcal{P}(\mathcal{P}_0)} \\
F_1 & : & \mathbf{ALG}_{\mathcal{P}_1} \rightarrow \mathbf{ALG}_{\mathcal{P}'(\mathcal{P}_1)} \\
F_2 & : & \mathbf{ALG}_{\mathcal{P}_2} \rightarrow \mathbf{ALG}_{\mathcal{P}''(\mathcal{P}_2)}
\end{array}
$$

$$
\begin{array}{ccccccc}
 & & & & \mathcal{P}_0 & \hookrightarrow & \mathcal{P}(\mathcal{P}_0) \\
 & & & & h \downarrow & (1) & \downarrow \\
 & & \mathcal{P}_1 & \hookrightarrow & \mathcal{P}'(\mathcal{P}_1) & \hookrightarrow & \mathcal{P} *_h \mathcal{P}'(\mathcal{P}_1) \\
 & & h' \downarrow & (2) & \hbar' \downarrow & (3) & \downarrow \\
\mathcal{P}_2 & \hookrightarrow & \mathcal{P}''(\mathcal{P}_2) & \hookrightarrow & \mathcal{P}' *_{h'} \mathcal{P}''(\mathcal{P}_2) & \hookrightarrow & (\mathcal{P} *_h \mathcal{P}') *_{\hbar'} (\mathcal{P}' *_{h'} \mathcal{P}'')(\mathcal{P}_2)
\end{array}
$$

Deux compositions de passages de paramètres génériques sont possibles, et elles conduisent à la même présentation paramétrée :

$$
(\mathcal{P}_2, (\mathcal{P} *_h \mathcal{P}') *_{h'} \mathcal{P}'') = (\mathcal{P}_2, \mathcal{P} *_{h' * h} (\mathcal{P}' *_{h'} \mathcal{P}'')) \qquad (1.4)
$$

avec $h' * h \overset{\text{déf}}{=} \hbar' \circ h$.

Rappelons que la composition de deux sommes amalgamées est une somme amalgamée (cf. §A.1.3.1). Par application de cette propriété à (1) et à (3), puis à (2) et à (3), on obtient que les diagrammes (1) $\cup$ (3) et (2) $\cup$ (3) représentent des sommes amalgamées. L'équation (1.4) est obtenue par combinaison de ces deux résultats.

À un isomorphisme naturel près, une relation semblable existe au niveau sémantique, entre les foncteurs libres :

$$
(F_0 *_h F_1) *_{h'} F_2 \cong F_0 *_{h' * h} (F_1 *_{h'} F_2)
$$

d'où l'on déduit :

$$
\mathrm{Ext}(\mathrm{Ext}(F_0, h) \circ F_1, h') \circ F_2 \cong \mathrm{Ext}(F_0, h' * h) \circ \mathrm{Ext}(F_1, h') \circ F_2
$$

Corollaire 1.21 ([58, 59]) *Le passage de paramètre standard appliqué à une composition de présentations paramétrées est équivalent à l'itération des passages de paramètres standards pour chaque présentation paramétrée.*

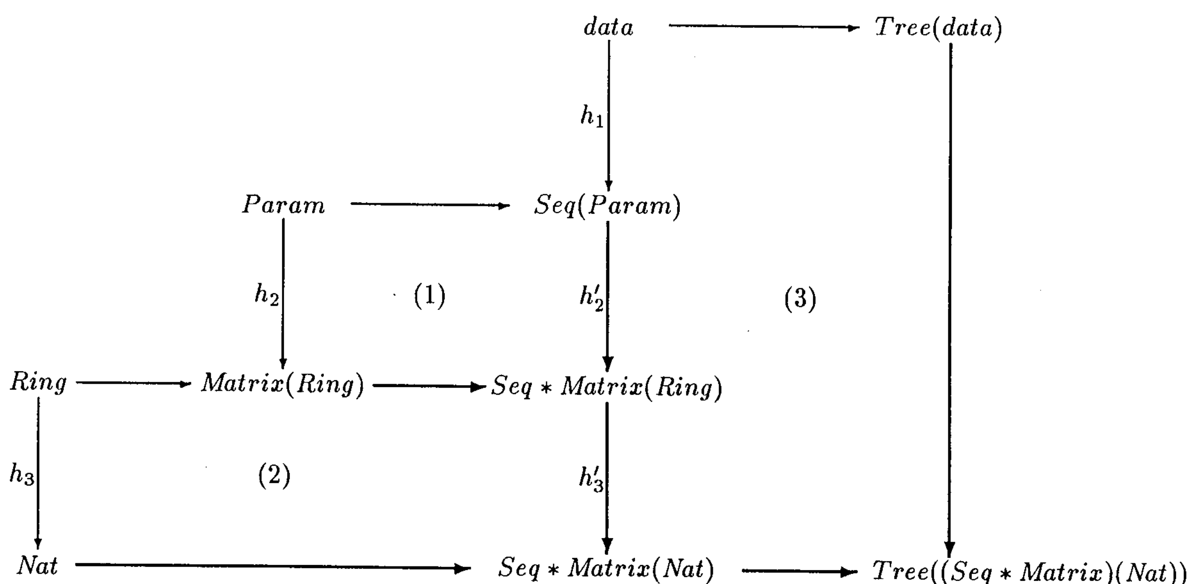
Ce corollaire découle de l'associativité de la composition des présentations paramétrées et de l'équivalence entre passage de paramètre standard et passage de paramètre générique avec $(\emptyset, \emptyset, \emptyset)$ comme présentation paramètre. Plus généralement, une stratégie d'évaluation de passages de paramètres successifs consiste en l'application, tant qu'elle est possible, de l'un des deux pas suivants :

paramétrisation application d'un passage de paramètre générique,

standard application d'un passage de paramètre standard.

Corollaire 1.22 ([58, 59]) *Le résultat d'une itération de passages de paramètres est indépendant de la stratégie d'évaluation.*

Nous reprenons, pour illustrer ce propos, l'exemple qui figure en [58, 59] :



Dans cet exemple, une stratégie "par valeur" revient à n'appliquer que des pas standards, pour donner, dans l'ordre :

$$\begin{array}{c} Matrix(Nat) \\ Seq(Matrix(Nat)) \\ Tree(Seq(Matrix(Nat))) \end{array}$$

tandis qu'une stratégie "par nom" consiste à appliquer d'abord des pas de paramétrisation et à construire :

$$\begin{array}{rcl} data & \longrightarrow & Tree * Seq(data) \\ Ring & \longrightarrow & ((Tree * Seq) * Matrix)(Ring) \\ & & ((Tree * Seq) * Matrix)(Nat) \end{array}$$

Le diagramme donne une stratégie "mixte" : un pas de paramétrisation (1) qui construit la présentation paramétrée $Ring \longrightarrow Seq * Matrix(Ring)$, puis deux pas standards ((2) et (3)) au moyen desquels on obtient respectivement $Seq * Matrix(Nat)$ et $Tree((Seq * Matrix)(Nat))$.

1.2 Généricité en FP2

1.2.1 Présentations en paramètre

En FP2, la description d'une présentation en paramètre s'effectue à l'aide d'une **propriété.** Une propriété est donc la donnée de sortes, d'opérateurs, et d'équations.

Exemple 1.23 *Propriété d'égalité.*

```
prop Equality[t / eq]
  opns eq      : t × t  →  Bool
  vars x, y, z : t
  eqns
    <>                                x eq x  ==  true
    <>                                x eq y  ==  y eq x
    <>  ((x eq y) and (y eq z)) ⇒ (x eq z)  ==  true
endprop
```

Une distinction importante, déjà soulignée dans [152], doit être opérée parmi les sortes et les opérateurs d'une propriété. FP2 permettant dans un module la visibilité des sortes et opérateurs définis dans les modules précédents, il est nécessaire de bien interpréter ces derniers comme dans la présentation qui les définit. Dans l'exemple 1.23, *Bool* doit être compris comme la sorte dont le support est égal à $\{true, false\}$, avec $true \not\equiv false$, et non une sorte destinée à paramétrer une unité générique. De plus, cette sorte ne doit pas être substituée par une autre sorte lors d'une instanciation de la propriété *Equality*, alors que les équations (1.2) et (1.3) autorisent le remplacement de *toutes* les sortes et de *tous* les opérateurs de la présentation paramètre : on voit ici apparaître des *contraintes* (d'après la terminologie de [152]) par rapport à la théorie exposée au §1.1.

Par contre, tout remplacement de t et d'eq effectué au moyen d'un morphisme de présentations est correct, et définit un modèle de la propriété *Equality*. Ce morphisme de présentations doit être étendu à l'identité pour les sortes et les opérateurs importés. Étant donné qu'une propriété représente avant tout un moyen de spécifier des présentations en paramètre, nous considérerons donc pour une propriété une sémantique lâche (cf. §1.1.2).

Par commodité, nous conviendrons que la *signature* d'une propriété est uniquement composée des sortes et opérateurs **non importés**, autrement dit, des sortes et des opérateurs **paramètres** qui ne doivent pas être interprétés initialement, et nous la citerons toujours dans le même ordre, exprimé dans l'en-tête, derrière la donnée du nom de la propriété :

$$Equality[t\ /\ eq]$$

Nous parlons ainsi de **signature ordonnée** d'une propriété — nous omettons les profils des opérateurs pour ne pas compliquer la notation. Les liaisons des morphismes de présentations seront données par l'utilisateur d'après cet ordre lexical, et, comme nous l'avons indiqué plus haut, le morphisme sera étendu à l'identité pour les sortes et opérateurs importés. (Nous verrons d'autres utilisations de cet ordre lexical au chapitre 3.) Plus généralement, nous appellerons **expression de modèle** toute expression composée d'un symbole de propriété, puis d'une image de sa signature ordonnée par un morphisme de présentations. Étant donné cette convention d'ordre lexical, une expression de modéle nous suffira pour spécifier une instanciation. Ainsi, l'expression :

$$Equality[Nat\ /\ =_N]$$

pourra représenter le morphisme $[t \mapsto Nat, eq \mapsto =_N]$. De même, le nom d'une entité générique, suivi d'une expression de modèle, suffira à désigner une **instance** de cette entité. Ainsi :

$$=_S.Equality[Nat \ / \ =_N]$$

Un autre point important est que les sortes et les opérateurs définis dans une propriété ne sont pas visibles à l'extérieur de la définition de cette propriété. En conséquence, lorsque nous considérerons plusieurs propriétés, ou plusieurs instances de la même propriété, leurs signatures seront toujours considérées comme **disjointes**.

1.2.2 Relations entre propriétés

Deux concepts principaux, l'**héritage** et la **satisfaction**, sont utilisés pour l'importation des définitions introduites au moyen de propriétés. Soient p et π des symboles de propriétés dont les signatures sont respectivement :

$$p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n] \text{ et } \pi[\tau_1, \ldots, \tau_u \ / \ \phi_1, \ldots, \phi_v]$$

Considérons la déclaration :

$$p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n] \ \mathbf{inherits} \ \pi[t'_1, \ldots, t'_u \ / \ f'_1, \ldots, f'_v]$$

Elle est licite s'il existe un morphisme de signature :

$$h = [\tau_1 \mapsto t'_1, \ldots, \tau_u \mapsto t'_u, \phi_1 \mapsto f'_1, \ldots, \phi_v \mapsto f'_n]$$

Alors, pour toute équation $(q_1 == q_2)$ de la présentation de π, $(h^\sharp(q_1) == h^\sharp(q_2))$ est importée dans la définition de p.

La déclaration :

$$p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n] \ \mathbf{satisfies} \ \pi[t'_1, \ldots, t'_u \ / \ f'_1, \ldots, f'_v]$$

est licite s'il existe un morphisme de signature :

$$h = [\tau_1 \mapsto t'_1, \ldots, \tau_u \mapsto t'_u, \phi_1 \mapsto f'_1, \ldots, \phi_v \mapsto f'_n]$$

tel que pour toute équation $(q_1 == q_2)$ de la présentation de π, $(h^\sharp(q_1) == h^\sharp(q_2))$ soit une conséquence logique des équations de p.

Autrement dit, l'héritage contient une information syntaxique qui sert à importer des équations, et la validation est alors immédiate, tandis que la satisfaction est une notion sémantique qui exprime que de tout modèle d'une propriété p, on peut extraire un modèle d'une propriété π. (Nous renvoyons à [53] pour des exemples de *validations* de telles déclarations dans le cadre de LPG[32].)

Exemple 1.24 *La propriété d'ordre partiel* satisfait *la propriété d'égalité sous-jacente (il est possible de démontrer que de tout modèle de* $Partial\text{-}Order[t_1 \ / \ rel_1, eq_1]$*, on peut extraire le modèle* $Equality[t_1 \ / \ eq_1]$*) ; l'ordre total* hérite *l'ordre partiel.*

[32] Ou plus précisément des exemples d'utilisation de l'outil de validation intégré à LPG.

prop *Partial-Order*$[t_1 \;/\; rel_1, eq_1]$
  opns $rel_1, eq_1 \;:\; t_1 \times t_1 \;\rightarrow\; Bool$
  vars $x, y, z \;:\; t_1$
  eqns
    <> $x \; rel_1 \; x \;==\; true$
    <> $(x \; rel_1 \; y) \; and \; (y \; rel_1 \; x) \;==\; x \; eq_1 \; y$
    <> $((x \; rel_1 \; y) \; and \; (y \; rel_1 \; z)) \Rightarrow (x \; rel_1 \; z) \;==\; true$
  satisfies *Equality*$[t_1 \;/\; eq_1]$
endprop

prop *Total-Order*$[t_2 \;/\; rel_2, eq_2]$
  opns $rel_2, eq_2 \;:\; t_2 \times t_2 \;\rightarrow\; Bool$
  vars $x, y \;:\; t_2$
  eqns
    <> $(x \; rel_2 \; y) \; or \; (y \; rel_2 \; x) \;==\; true$
  inherits *Partial-Order*$[t_2 \;/\; rel_2, eq_2]$
endprop

Ces deux déclarations sont présentes en LPG. Pour des raisons de simplicité, elles ont été fusionnées en une déclaration "**includes**" en FP2, que l'on doit comprendre comme une déclaration d'héritage, puisque la notion d'héritage inclut celle de satisfaction. Les déclarations :

$$p[t_1, \ldots, t_m \;/\; f_1, \ldots, f_n] \; \textbf{includes} \; \pi[t'_1, \ldots, t'_u \;/\; f'_1, \ldots, f'_v]$$

induisent une relation dont la fermeture réflexive et transitive, que nous noterons "$\supseteq$", est l'inclusion entre expressions de modèles :

$$p[t_1, \ldots, t_m \;/\; f_1, \ldots, f_n] \supseteq \pi[t'_1, \ldots, t'_u \;/\; f'_1, \ldots, f'_v]$$
$$\overset{\text{déf}}{\Longleftrightarrow}$$
$$p[t_1, \ldots, t_m \;/\; f_1, \ldots, f_n] \; \textbf{includes} \ldots \textbf{includes} \; \pi[t'_1, \ldots, t'_u \;/\; f'_1, \ldots, f'_v]$$

Bien évidemment, pour toute expression de modèle $p[t_1, \ldots, t_m \;/\; f_1, \ldots, f_n]$, on a :

$$p[t_1, \ldots, t_m \;/\; f_1, \ldots, f_n] \supseteq \diamond$$

Note La version 1.8 de LPG [19] offre également une directive **combines**, qui sert à exprimer qu'une propriété est la réunion d'autres propriétés. Par exemple :

prop $Ftype^2[t_1, t_2 \;/\;]$ **combines** $Ftype[t_1 \;/\;], Ftype[t_2 \;/\;]$ -- Cf. exemple 0.83.
endprop

Cette construction impose qu'il n'y ait aucune équation dans le corps de la propriété que l'on définit, et que chaque sorte ou opérateur de la signature de cette propriété figure dans au moins une signature des propriétés à combiner.

1.2.3 Présentations paramétrées

La présentation paramètre est spécifiée par une propriété, appelée **propriété exigée**, et la présentation cible est un module de type ou d'enrichissement. Les sortes et les opérateurs définis dans une propriété n'étant pas visibles à l'extérieur, le morphisme de présentations qui relie la propriété au

module générique est un renommage bijectif, pas nécessairement l'injection canonique. Ce renommage bijectif est lui aussi spécifié lexicalement, d'après la signature ordonnée de la propriété. Nous appelons **modèle exigé** l'expression de modèle qui regroupe la propriété exigée et les sortes et opérateurs utilisés dans le corps de la présentation cible. Ces sortes et opérateurs sont dits **formels**.

Exemple 1.25 *Ordre lexicographique sur les séquences génériques. La sorte Seq est supposée* importée *d'après une définition précédente (cf. exemple 0.87).*

enr req *Total-Order*[*t* / *rel*, *eq*]
  opns $\leqslant$: $Seq[t] \times Seq[t] \rightarrow Bool$
  vars x_1, x_2 : t
    s_1, s_2 : $Seq[t]$
  rules
    <> $nil \leqslant s_2$ ==> *true*
    <> $(x_1$ <+ $s_1) \leqslant nil$ ==> *false*
    <> $(x_1$ <+ $s_1) \leqslant (x_2$ <+ $s_2)$ ==> **if** x_1 *eq* x_2 **then** $s_1 \leqslant s_2$ **else** x_1 *rel* x_2
      endif
endenr

Le renommage bijectif reliant cet enrichissement à la propriété *Total-Order* est :

$$[t_2 \mapsto t, rel_2 \mapsto rel, eq_2 \mapsto eq]$$

Une caractéristique importante de LPG et de FP2 est que l'instanciation des unités génériques est *locale*, donnée pour chaque occurrence de la sorte ou de l'opérateur générique. Dans le passage de paramètre donné en [58], on remarquera que les sortes et opérateurs de la présentation cible sont simplement *importés*. Dans le contexte de l'instanciation, ils représentent les sortes et opérateurs résultats de celle-ci.

En LPG et en FP2, chaque sorte ou opérateur est soit formel, c'est-à-dire présent dans la signature du modèle exigé, soit paramétré par une expression de modèle. Une simplification : si l'expression de modèle qui paramètre une sorte ou un opérateur est la même que le modèle exigé, sa mention est facultative. Ainsi, dans la dernière équation de l'exemple 1.25 :

$(x_1$ <+ $s_1) \leqslant (x_2$ <+ $s_2)$ ==> **if** x_1 *eq* x_2 **then** $s_1 \leqslant s_2$ **else** x_1 *rel* x_2
endif

"$\leqslant$" représente "$\leqslant$.*Total-Order*[*t* / *rel*, *eq*]".

Par homogénéisation, nous considérons que les sortes et opérateurs non paramétrés le sont en fait par une expression de modèle "vide" sans sortes, ni opérateurs, c'est-à-dire un modèle de la présentation "$\diamond$".

À noter que les types étant protégés en FP2 par l'impossibilité d'ajouter de nouveaux constructeurs ailleurs que dans le module qui les définit, un module générique (module de type ou d'enrichissement) n'ajoute pas de nouveaux termes dont la sorte apparaît dans la signature de la propriété, les opérateurs définis dans la présentation cible, et dont le codomaine est une sorte formelle de la propriété exigée ne pouvant être constructeurs (voir le traitement des opérateurs non complètement définis au chapitre 4). De même, l'impossibilité d'équations entre constructeurs empêche une validation lorsque, pour une sorte, les équations d'une propriété perturbent la congruence entre les termes engendrés uniquement à l'aide de constructeurs. Par conséquent, on peut considérer que les foncteurs qui sont sémantiques des présentations paramétrées de FP2 sont fortement persistants.

1.2.4 Morphismes de présentations

Un morphisme de présentations s'exprime en FP2 par un module de **modèle**. Comme annoncé au §1.2.1, l'ordre suivi est l'ordre lexical de la signature ordonnée.

model $INCNAT$ **is** $Total\text{-}Order[Nat\ /\ \leqslant_N, =_N]$
endmodel

Le morphisme est : $[t \mapsto Nat, rel \mapsto \leqslant_N, eq \mapsto =_N]$.

Remarque Les implantations réalisées de LPG et de FP2 permettent la substitution d'opérateurs formels par des variables, une variable de sorte s pouvant dans ce cas être assimilée à un opérateur de profil "$- \rightarrow s$" — cf. exemple B.8. Cette facilité, en fait largement utilisée dans [9], n'a malheureusement pas actuellement — pas encore ? — de fondement sémantique. Ajoutons qu'au niveau des applications réalisées, l'utilisation de cette possibilité constitue malgré tout une justification supplémentaire à un mécanisme d'instanciation local.

1.2.5 Modèles génériques

D'après les définitions et résultats du §1.1, nous pouvons par exemple spécifier le type des séquences génériques. Si, dans la présentation paramètre, est spécifiée une relation d'ordre sur la sorte t, nous pouvons également (cf. exemple 1.25) définir l'ordre lexicographique sur les séquences génériques, chaque type séquence étant dès lors muni d'une relation d'ordre total. Il est par conséquent possible d'ordonner lexicographiquement les séquences de séquences, l'instanciation procédant par composition de passage de paramètres.

Ce que nous aimerions pouvoir réaliser, c'est, à partir d'un morphisme de présentation dont la source est la propriété d'ordre total sur les éléments de t, obtenir directement le morphisme dont la source est l'expression de modèle représentant l'ordre total sur les séquences de t.

En FP2, une telle possibilité est offerte par l'introduction des modèles génériques. Considérons p_0 et p deux symboles de propriétés dont les signatures ordonnées, supposées disjointes, sont respectivement :

$$p_0[t_1, \ldots, t_m\ /\ f_1, \ldots, f_n] \text{ et } p[\tau_1, \ldots, \tau_u\ /\ \phi_1, \ldots, \phi_v]$$

Une déclaration de **modèle générique** a la forme :

model M **req** $p_0[t_1, \ldots, t_m\ /\ f_1, \ldots, f_n]$ **is** $p[t'_1, \ldots, t'_u\ /\ f'_1, \ldots, f'_n]$
endmodel

Elle signifie qu'à tout morphisme de présentations :

$$h = [t_1 \mapsto t_{0,1}, \ldots, t_m \mapsto t_{0,m}, f_1 \mapsto f_{0,1}, \ldots, f_n \mapsto f_{0,n}]$$

on peut associer un autre morphisme de présentations :

$$h' = [\tau_1 \mapsto t'_1, \ldots, \tau_u \mapsto t'_u, \phi_1 \mapsto f'_1, \ldots, \phi_v \mapsto f'_v]$$

tel que si h est valide, alors $h \circ h'$ est valide.

Si M_0 est une déclaration de modèle représentant un morphisme de présentations $\hbar$ de source p_0, alors $M.M_0$ représente le morphisme $\hbar \circ h'$.

Exemple 1.26 *Soit le modèle générique :*

model $ORDSEQ$ **req** $Total\text{-}Order[t \;/\; rel, eq]$ **is**
$Total\text{-}Order[Seq[t] \;/\; \leqslant_S.Total\text{-}Order[t \;/\; rel, eq], =_S.Equality[t \;/\; eq]]$
endmodel

Soit un morphisme de présentations quelconque appliqué à $Total\text{-}Order[t \;/\; rel, eq]$, il existe un morphisme de présentations, appliqué à $Total\text{-}Order[t' \;/\; rel', eq']$, qui est :

$$\begin{array}{rcl} t' & \mapsto & Seq[t] \\ rel' & \mapsto & \leqslant_S.Total\text{-}Order[t \;/\; rel, eq] \\ eq' & \mapsto & =_S.Equality[t \;/\; eq] \end{array}$$

Soit le morphisme $[t \mapsto Nat, rel \mapsto \leqslant_N, eq \mapsto =_N]$, appliqué à $Total\text{-}Order[t \;/\; rel, eq]$, et qui peut être décrit par l'expression de modèle $Total\text{-}Order[Nat \;/\; \leqslant_N, =_N]$. Alors l'expression $ORDSEQ.Total\text{-}Order[Nat \;/\; \leqslant_N, =_N]$ représente le morphisme :

$$\begin{array}{rcl} t' & \mapsto & Seq[Nat] \\ rel' & \mapsto & \leqslant_S.Total\text{-}Order[Nat \;/\; \leqslant_N, =_N] \\ eq' & \mapsto & =_S.Equality[Nat \;/\; =_N] \end{array}$$

La composition de morphismes étant associative, il est à noter que pour tous modèles génériques M, M_0, M_1, le modèle $M.(M_1.M_0)$ représente le même morphisme que le modèle $(M.M_1).M_0$.

1.2.6 Passage de paramètres

La notion de base pour l'instanciation en FP2 est celle de couple d'instanciation. Un **couple d'instanciation** est un couple d'expressions de modèles, noté :

$$\langle \mu_1, \mu_2 \rangle$$

tels que les paramètres formels de μ_1 incluent ceux de μ_2. Un couple d'instanciation considéré globalement n'a donc pas de paramètres. Un objet (sorte ou opérateur) exigeant le modèle μ_0 peut être instancié par $\langle \mu_1, \mu_2 \rangle$ sous le modèle exigé RM s'il existe un morphisme de présentations $h : \mu_0 \rightarrow \mu_2$ et si $RM \supseteq \mu_1$.

Le modèle exigé RM est soit celui du module (type, enrichissement, propriété, modèle, ou processus) en cours de traitement, soit "$\diamond$" par défaut (par exemple, si l'on se trouve à l'intérieur de la boucle d'évaluation d'expressions).

μ_2 contient les paramètres effectifs, et μ_1 est le modèle exigé local. Il est nécessaire que tout modèle de la présentation décrite par le modèle exigé RM soit un modèle de la présentation décrite par μ_1, d'où la condition $RM \supseteq \mu_1$. Cette notion de couple d'instanciation permet de traiter dans un même moule l'instanciation complète d'une unité générique, le renommage des paramètres d'une propriété, et certains cas d'instanciations partielles.

Exemple 1.27 (Instanciation complète) $\langle \diamond, Total\text{-}Order[Bool \;/\; \Rightarrow, \Leftrightarrow] \rangle$ *est un couple d'instanciation de la propriété* $Total\text{-}Order$ *dont la signature ordonnée est "*$[t \;/\; rel, eq]$*".*

Exemple 1.28 (Renommage des paramètres)

- *Sous le modèle exigé* $Total\text{-}Order[t_1 \ / \ rel_1, eq_1]$,

$$\langle Total\text{-}Order[t_1 \ / \ rel_1, eq_1], Total\text{-}Order[t_1 \ / \ rel_1, eq_1]\rangle$$

est un couple d'instanciation de $Total\text{-}Order[t \ / \ rel, eq]$. *Le morphisme de présentations est* $[t \mapsto t_1, rel \mapsto rel_1, eq \mapsto eq_1]$.

- *Sous le modèle exigé* $Total\text{-}Order[t_1 \ / \ rel_1, eq_1]$,

$$\langle Equality[t_1 \ / \ eq_1], Equality[t_1 \ / \ eq_1]\rangle$$

est un couple d'instanciation de $Equality[t \ / \ eq]$. *Le morphisme de présentations est* $[t \mapsto t_1, eq \mapsto eq_1]$, *et l'on a* $Total\text{-}Order[t_1 \ / \ rel_1, eq_1] \supseteq Equality[t_1 \ / \ eq_1]$ *— cf. exemple 1.24.*

Exemple 1.29 (Instanciation partielle) *Soit la propriété* $Ftype^3$ *suivante :*

prop $Ftype^3[t_1, t_2, t_3 \ / \ ]$
  includes $Ftype[t_1 \ / \ ], Ftype[t_2 \ / \ ], Ftype[t_3 \ / \ ]$
endprop

Sous le modèle exigé $Ftype[t \ / \ ]$, $\langle Ftype[t \ / \ ], Ftype^3[Nat, t, Nat \ / \ ]\rangle$ *est un couple d'instanciation de* $Ftype^3[t_1, t_2, t_3 \ / \ ]$. *Le morphisme de présentation est* $[t_1 \mapsto Nat, t_2 \mapsto t, t_3 \mapsto Nat]$, *et tout modèle de* $Ftype[t \ / \ ]$ *est un modèle de* $Ftype^3[Nat, t, Nat \ / \ ]$.

Les conditions que nous venons de donner sont purement *sémantiques*. En pratique, l'analyseur FP2 n'effectue que les contrôles sur les sortes et les profils des opérateurs, et c'est à l'utilisateur d'assumer que les équations sont satisfaites. (Dans la version 1.8 de LPG, un outil de validation permet d'effectuer de telles vérifications de façon guidée [53].)

Un point important est que la mention par l'utilisateur des deux membres du couple d'instanciation n'est pas nécessaire : l'analyseur FP2 reconstruit le premier membre en prenant une expression de modèle minimale par rapport à l'ordre partiel sur les propriétés.

(1) Si le second membre ne contient pas de paramètres formels, le premier membre est "$\diamond$".

(2)
- Si le modèle exigé est égal, à un renommage près, au second membre,
- ou s'il inclut une expression de modèle égale, à un renommage près, au second membre,

alors on prend comme premier membre le second membre.

(3) Si la propriété exigée du second membre ne contient pas d'équations, et si chacune des inclusions induites par le second membre est :

- soit une expression de modèle licite,
- soit incluse dans le modèle exigé,

alors on prend comme premier membre le modèle exigé.

On remarque que les cas (1) et (2) s'appliquent sans difficulté aux exemples 1.27 et 1.28. Le cas (3) sert à traiter certaines instanciations partielles, comme nous allons le voir ci-dessous (voir aussi l'exemple B.11).

Exemple 1.30 (Suite de l'exemple 1.29) *Soit l'instanciation* $Ftype^3[Nat, t, Nat\ /\ ]$ *sous le modèle exigé* $Ftype[t\ /\ ]$*. Par construction, la propriété* $Ftype^3$ *vérifie les inclusions suivantes :*

$$\begin{aligned} Ftype^3[t_1, t_2, t_3\ /\ ] &\supseteq Ftype[t_1\ /\ ] \\ &\supseteq Ftype[t_2\ /\ ] \\ &\supseteq Ftype[t_3\ /\ ] \end{aligned} \tag{1.5}$$

D'une part, le type Nat est un modèle de la propriété Ftype, d'autre part, nous avons par (1.5) l'inclusion des deux expressions de modèles :

$$Ftype^3[Nat, t, Nat\ /\ ] \supseteq Ftype[t\ /\ ]$$

d'où l'on déduit le couple d'instanciation :

$$\langle Ftype[t\ /\ ], Ftype^3[Nat, t, Nat\ /\ ]\rangle$$

Note Dans la maquette préparatoire à FP2 [105], il existait une possibilité d'*hériter* une définition de propriété à l'intérieur d'un module de type ou d'enrichissement. Ce mécanisme, exactement semblable à l'héritage entre propriétés, permettait, au niveau opérationnel, de spécifier des règles de réécriture dans des propriétés, et de les importer dans un module de type ou d'enrichissement.

1.2.7 Sémantique opérationnelle

Après avoir fixé les conditions de FP2 en ce qui concerne le passage de paramètres, nous allons à présent donner une sémantique opérationnelle précise des opérateurs génériques. Nous supposons que les termes sont sémantiquement corrects, c'est-à-dire que chaque opérateur générique est qualifié par une expression correcte de modèle (nous verrons au chapitre 3 comment l'utilisateur peut parfois sous-entendre tout ou partie de cette indication). Pour évaluer les opérateurs non constructeurs, nous avons à disposition un système de réécriture R convergent et une fonction $rewrite_R$ qui applique un pas de réécriture à son argument. Les autres notations — éventuellement indicées — sont :

- ϕ désigne un opérateur formel quelconque ;
- p' est un nom de propriété ;
- f désigne un opérateur générique quelconque, constructeur ou non ;
- c désigne un opérateur générique constructeur quelconque ;
- g désigne un opérateur générique non constructeur quelconque, dont le modèle exigé est $p[\tau_1, \ldots, \tau_u\ /\ \phi_1, \ldots, \phi_v]$;
- t désigne une expression quelconque de sorte ;
- $\tilde{f}$ désigne soit un opérateur formel, soit un opérateur générique donné avec l'expression de modèle qui le qualifie ;
- ρ est l'environnement générique : il contient les liaisons des paramètres formels aux paramètres effectifs.

En ce qui concerne l'environnement ρ, nous supposons en sus que les conflits de noms sont résolus (éventuellement par des renommages successifs). Voici maintenant la définition de la fonction $eval_{FP2}$ qui, étant donné, d'une part, un terme complètement instancié construit avec un opérateur générique, d'autre part, un environnement ρ, renvoie la forme normale de ce terme.

$$eval_{FP2}(c.M(x_1,\ldots,x_n))(\rho) = c(eval_{FP2}(x_1)(\rho),\ldots,eval_{FP2}(x_n)(\rho)) \quad (1.6)$$

$$eval_{FP2}(g.\Diamond(x_1,\ldots,x_n))(\rho) = eval_{FP2}(rewrite_R(g(eval_{FP2}(x_1)(\rho),\ldots,eval_{FP2}(x_n)(\rho))))(\rho) \quad (1.7)$$

$$eval_{FP2}(g.p[t_1,\ldots,t_u \ / \ \tilde{f}_1,\ldots,\tilde{f}_v](x_1,\ldots,x_n))(\rho) = eval_{FP2}(rewrite_R(g(eval_{FP2}(x_1)(\rho),\ldots,eval_{FP2}(x_n)(\rho)))) (\rho \cup \{[\tau_i \mapsto t_i]_{1\leqslant i\leqslant u}\} \cup \{[\phi_i \mapsto \tilde{f}_i]_{1\leqslant i\leqslant v}\}) \quad (1.8)$$

$$eval_{FP2}(\phi_j(x_1,\ldots,x_{n'}))(\rho \uplus ([\phi_j \mapsto f_j.p'[t'_1,\ldots,t'_{u'} \ / \ \tilde{f}'_1,\ldots,\tilde{f}'_{v'}]])) = eval_{FP2}(f_j.p'[t'_1,\ldots,t'_{u'} \ / \ \tilde{f}'_1,\ldots,\tilde{f}'_{v'}](x_1,\ldots,x_{n'}))(\rho) \quad (1.9)$$

$$eval_{FP2}(\phi_j(x_1,\ldots,x_{n'}))(\rho \uplus ([\phi_j \mapsto \phi'_j])) = eval_{FP2}(\phi'_j(x_1,\ldots,x_{n'}))(\rho) \quad (1.10)$$

$$eval_{FP2}(\mathbf{if}\ x_1\ \mathbf{then}\ x_2\ \mathbf{else}\ x_3\ \mathbf{endif})(\rho) = \mathbf{if}\ eval_{FP2}(x_1)(\rho) = true\ \mathbf{then}\ eval_{FP2}(x_2)(\rho)\ \mathbf{else}\ eval_{FP2}(x_3)(\rho)\ \mathbf{endif}$$

Comme on a pu le remarquer, nous avons indiqué dans les règles (1.6) et (1.7) que le mécanisme d'évaluation employé est l'appel par valeur. La règle (1.6) traduit que l'évaluation d'un terme formé par un opérateur constructeur qui est appliqué à des arguments en forme normale est le terme lui-même[33]. La règle (1.7) donne le cas du modèle exigé "◇" : remarquons qu'elle est en réalité un cas particulier de la règle (1.8). Enfin, le terme étant supposé complètement instancié, la règle (1.9) indique comment un opérateur formel est *remplacé* par l'opérateur effectif qui lui correspond dans l'environnement générique. Le cas d'un opérateur formel lié à un autre opérateur formel est abordé par la règle (1.10), similaire à la règle (1.9). Le terme à évaluer étant par hypothèse complètement instancié, tout opérateur formel est instancié à un opérateur effectif par fermeture transitive des liaisons présentes dans l'environnement ρ.

Exemple 1.31 *Évaluation d'une expression d'égalité entre objets de sorte Seq[Seq[Seq[Nat]]].*

$$=_S.Equality[Seq[Seq[Nat]] \ / \ =_S.Equality[Seq[Nat] \ / \ =_S.Equality[Nat \ / \ =_N]]]([[[0]]],[[[0]]])$$

==>

if $=_S.Equality[Seq[Nat] \ / \ =_S.Equality[Nat \ / \ =_N]]([[0]],[[0]])$ **then** $=_S.\cdots(nil,nil)$ **else** *false* **endif**

==>

[33]Nous ne nous préoccupons pas ici des constructeurs partiels, qui seront abordés au chapitre 4.

if **if** $=_S.Equality[Nat\ /\ =_N]([0],[0])$ **then** $=_S.\cdots(nil, nil)$ **else** *false*
  endif
  then $=_S.\cdots(nil, nil)$ **else** *false*
endif

==>

if **if** **if** $0 =_N 0$ **then** $=_S.\cdots(nil, nil)$ **else** *false*
    endif
    then $=_S.\cdots(nil, nil)$ **else** *false*
  endif
  then $=_S.\cdots(nil, nil)$ **else** *false*
endif

==>

true

1.3 Généricité en OBJ3

La généricité en OBJ3 est fondée sur les mêmes principes sémantiques, inspirés de CLEAR [34], et de [58, 59], que l'on retrouve en LPG et en FP2. Les différences se situent au niveau de la mise en œuvre, de la gestion des noms, et des raccourcis qui permettent d'exprimer les morphismes de théories (nous les verrons au §3.7). Le lecteur pourra comparer les exemples donnés avec les spécifications correspondantes en FP2, données dans les chapitres 0, 1 et 3 ou à l'annexe B.

1.3.1 Théories

Un module de **théorie** est à OBJ3 ce qu'est un module de propriété à FP2 : il décrit une présentation en paramètre.

Exemple 1.32 *Théorie comprenant une sorte formelle.*

```
th TRIV is sort Elt .
ht
```

Les directives pour l'importation de modules (cf. §0.1.4.3) s'appliquent aux théories. Ainsi, la théorie des ensembles partiellement ordonnés nécessite et protège l'objet `BOOL`. La théorie des ensembles totalement ordonnés est vue comme une extension de celle des ensembles partiellement ordonnés. Plus généralement, la directive **using** d'OBJ3 peut être employée là où une directive **includes** serait utilisée en FP2.

Exemple 1.33 *Théories des ensembles partiellement et totalement ordonnés.*

```
th POSET is
   protecting BOOL .
   sort Elt .
   op   _<_       : Elt Elt -> Bool .
   vars e1 e2 e3 : Elt .
   eq                                e1 < e1 = false .
```

```
   eq   ((e1 < e2) and (e2 < e3)) implies (e1 < e3) = true .
ht

th TOSET is
   using POSET .
   vars e1 e2 : Elt .
   eq   (e1 < e2) or (e2 < e1) or (e1 == e2) = true .
ht
```

Note "==" désigne en OBJ3 l'égalité syntaxique entre termes. La négation de cet opérateur est notée "=/=".

Bien sûr, le même problème qu'en FP2 se pose quant à l'interprétation initiale des sortes et des opérateurs importés. D'après notre terminologie, la signature de la théorie TOSET est :

```
(Elt,< : Elt Elt -> Bool)
```

1.3.2 Objets paramétrés

Un objet peut être paramétré par *plusieurs* théories. Chaque théorie utilisée comme présentation paramètre est désignée par une *étiquette*. Les noms visibles dans la théorie considérée sont alors visibles dans la spécification de l'objet paramétré, indicés par l'étiquette de la théorie.

Exemple 1.34 *Produit cartésien générique de deux types.*

```
obj 2TUPLE[X1 :: TRIV, X2 :: TRIV] is
   sort 2Tuple .
   op  <<_;_>> : Elt.X1 Elt.X2 -> 2Tuple .
   op  1*_     : 2Tuple        -> Elt.X1 .
   op  2*_     : 2Tuple        -> Elt.X2 .
   var e1      : Elt.X1 .
   var e2      : Elt.X2 .
   eq  1* << e1 ; e2 >> = e1 .
   eq  2* << e1 ; e2 >> = e2 .
jbo
```

Exemple 1.35 *Ordre lexicographique sur le type 2Tuple.*

```
obj LEXICO[X1 :: TOSET, X2 :: TOSET] is
   protecting 2TUPLE[X1,X2] .
   op   _<_    : 2Tuple 2Tuple -> Bool .
   vars e1 e'1 : Elt.X1 .
   vars e2 e'2 : Elt.X2 .
   eq   << e1 ; e2 >> < << e'1 ; e'2 >> = if e1 == e'1 then e2 < e'2 else e1 < e'1
                                           fi .
jbo
```

Bien sûr, dans le cas d'une seule théorie paramètre, la mention de l'étiquette pour les noms importés n'est pas nécessaire. Autrement dit, pour une présentation paramétrée $\mathcal{P}_0 \xrightarrow{s} \mathcal{P}$, s est l'injection canonique en OBJ3 (rappelons que c'est un renommage bijectif en FP2).

Exemple 1.36 *Type des séquences linéaires.*

```
obj SEQ[X :: TRIV] is
   sorts NeSeq Seq .
   subsorts NeSeq < Seq .
   op   nil     :            -> Seq .
   op   __      : Elt   Seq -> NeSeq .
   op   head_   : NeSeq      -> Elt .
   op   tail_   : NeSeq      -> Seq .
   op   _$_     : Seq   Seq -> Seq .    --- concatenation.
   op   _$_     : NeSeq Seq -> NeSeq .
   var  e       : Elt .
   vars sq sq0 : Seq .
   eq    head (e sq) = e .
   eq    tail (e sq) = sq .
   eq       nil $ sq0 = sq0 .
   eq   (e sq) $ sq0 = e (sq $ sq0) .
jbo
```

Exemple 1.37 *Type générique des arbres binaires ordonnées.*

```
obj OTREE[X :: TOSET] is
   protecting SEQ[X] .
   sorts NeOTree OTree .
   subsorts NeOTree < OTree .
   op   onil      :                     -> OTree .
   op   _<#_#>_   : OTree Elt OTree    -> NeOTree .
   op   _ins_     :       Elt OTree    -> NeOTree .
   op   lin_      :           NeOTree -> NeSeq .
   op   lin_      :           OTree    -> Seq .
   vars e1 e2     : Elt .
   vars t1 t2 t3 : OTree .
   ceq  (t1 <# e1 #> t2) <# e2 #> t3 = (t1 <# e2 #> t2) <# e1 #> t3 if e2 < e1 .
   ceq  t1 <# e1 #> (t2 <# e2 #> t3) = t1 <# e2 #> (t2 <# e1 #> t3) if e2 < e1 .
   eq                      e1 ins t1 = t1 <# e1 #> onil .
   eq                       lin onil = nil .
   eq            lin (t1 <# e1 #> t2) = (lin t1) $ (e1 (lin t2)) .
jbo
```

À noter que le foncteur libre d'une présentation paramétrée d'OBJ3 n'est pas nécessairement persistant, ainsi que le montre l'exemple suivant, tout à fait licite :

Exemple 1.38 *Ajout à un type d'un élément minimum et d'un élément maximum (cf. exemple 1.10).*

```
obj LATTICE[X :: POSET] is
   op   bottom : -> Elt .
   op   top    : -> Elt .
```

```
   vars x        : Elt .
   eq   bottom < bottom = false .
   eq   top    < top    = false .
   eq   bottom < top    = true .
   ceq  bottom < x      = true if x =/= bottom .
   ceq  x      < top    = true if x =/= top .
jbo
```

1.3.3 Morphismes de présentation

Spécifier un morphisme d'une théorie vers un objet s'effectue grâce à une **vue** (*view*). À noter que ce morphisme doit être compatible avec l'ordre entre les sortes. La notion de vue paramétrée — qui pourrait correspondre aux modèles génériques de FP2 — n'existe pas en OBJ3, elle a été néanmoins envisagée dans [73].

Exemple 1.39 *Arbres binaires d'entiers, ordre décroissant.*

```
view DECNAT of NAT as TOSET is
   sort Elt to Nat .
   op _<_ to _>_ .
weiv

obj OTDECNAT is protecting OTREE[DECNAT] .
jbo
```

L'instanciation d'un module étant globale, un identificateur (par exemple, `Seq`, et non une expression *Seq*[...], comme en FP2) suffit pour désigner une sorte ou un opérateur provenant d'un module instancié, et cet identificateur doit être compris dans le contexte de l'instanciation. Ainsi, après la directive "`protecting SEQ[NAT]`", `Seq` désigne les séquences d'entiers naturels. On évite par *renommage* les conflits de noms provenant d'importations avec une instanciation différente. (Le même procédé est nécessaire pour la même raison en ACT-ONE [59].)

Exemple 1.40 *Utilisation à l'intérieur d'un même module de séquences d'entiers et de séquences de booléens.*

```
obj SWITCH is
   protecting SEQ[NAT]  * (sort Seq to Seqnat) .
   protecting SEQ[BOOL] * (sort Seq to Seqbool) .
   op  f  : Seqnat Seqbool -> Nat .
   var xn : Nat .
   var xb : Bool .
   var sn : Seqnat .
   var sb : Seqbool .
   eq      f(nil,nil) = 0 .
   eq    f(nil,xb sb) = 1 .
   eq    f(xn sn,nil) = 2 .
   eq  f(xn sn,xb sb) = 3 .
jbo
```

Exemple 1.41 *Exemple récapitulatif des piles bornées, avec utilisation d'une contrainte de sorte (cf. §0.1.2.5).*

```
th NAT* is
   protecting NAT .
   op bound : -> Nat .
ht

obj BOUNDED-STACK [X :: TRIV,Y :: NAT*] is
   sorts NeBStack BStack ErrBStack .
   subsorts NeBStack < BStack < ErrBStack .
   op     empty    :                       -> BStack .
   op     push     : Elt      ErrBStack -> ErrBStack .
   op     top_     : NeBStack              -> Elt .
   op     pop_     : NeBStack              -> BStack .
   op     length_  : BStack                -> Nat .
   var    u        : Elt .
   var    bs       : BStack .
   op-as push      : Elt BStack -> NeBStack for push(u,bs) if length(bs) < bound .
   eq        top push(u,bs) = u .
   eq        pop push(u,bs) = bs .
   eq          length empty = 0 .
   eq     length push(u,bs) = s length bs .
jbo
```

Note Le lecteur intéressé trouvera en [75] une spécification de constructions géométriques utilisant le formalisme algébrique d'OBJ3. Y figurent des exemples de théories (parmi lesquelles la théorie des corps, donnée à l'aide d'une contrainte de sorte) et d'objets paramétrés.

1.3.4 Comparaison

La différence majeure réside dans le fait que l'instanciation d'une unité générique s'effectue *globalement* en OBJ3, par instanciation de tout un module et importation de ce module, alors qu'elle s'effectue *localement* en LPG et en FP2, où chaque sorte, chaque opérateur possède une instanciation locale compatible avec le modèle exigé. Les conventions d'OBJ3 sont plus proches de la théorie, alors que celles de FP2, par leur utilisation d'expressions de sortes et d'opérateurs, sont peut-être plus didactiques. Enfin, pour les mêmes raisons, la gestion des noms pour la signature d'une propriété et les paramètres formels d'un module paramétré nous semble plus claire en FP2.

Si les primitives d'importation d'OBJ3 permettent une plus grande liberté dans l'utilisation des objets, et une meilleure mise en évidence du graphe de dépendance des définitions — ce que n'offre pas FP2 —, par contre, nous trouvons moins claire l'utilisation de ces mêmes primitives pour l'importation entre spécifications de théories, et nous préférons la formulation des relations entre propriétés par les clauses "hérite" et "satisfait", qui nous semble plus lisible, tant du point de vue de l'utilisateur que du point de vue sémantique.

Chapitre 2

Réalisation

Dans une implantation réelle d'un langage fondé sur la réécriture, il est possible d'*exécuter directement* cette réécriture — une telle solution a été choisie dans OBJ2 et OBJ3[34] —, il est également possible de *compiler* chaque fonction définie au moyen de règles de réécriture, le résultat étant un code écrit en un langage cible — c'est la démarche adoptée en HOPE [35], où le langage cible est un code pour machine abstraite, et en LPG (des exemples de codes générés sont donnés en [20]). FP2 a choisi la seconde voie : toute définition d'opérateur FP2 est compilée en une fonction Common Lisp[35].

En fait, nous n'utilisons qu'un sous-ensemble purement fonctionnel de Common Lisp. Il serait parfaitement possible de décrire notre procédé de compilation dans n'importe quel langage fonctionnel (λ-calcul), même typé (par exemple, ML [89]). Seules des raisons de commodité de notation[36] nous font utiliser Common Lisp dans la suite de l'exposé. Au niveau de la formalisation, choisir un langage fonctionnel comme langage cible d'une opération de compilation apporte le principal avantage de faciliter grandement la preuve de celle-ci. Une telle démarche, mais poussée beaucoup plus loin vers un langage cible fonctionnel simulant un code pour une machine virtuelle se retrouve dans [61, 62] : le but est alors d'expliciter *entièrement* l'opération de compilation en restant dans un cadre fonctionnel.

Nous commençons d'abord par traiter des principes généraux qui guident la compilation des objets de la partie fonctionnelle : sortes et opérateurs. Nous résumons dans un tableau (p. 86) les choix de représentation effectués lors de la compilation des sortes. En ce qui concerne la compilation des systèmes de réécriture, nous ne l'abordons pas en profondeur dans le cadre de FP2, renvoyant le lecteur intéressé aux ouvrages de base [28, 127, 101]. Après un bref rappel du principe de l'algorithme de compilation des parties gauches, décrit dans [164], nous exposons en détail le principal apport de ce chapitre : la compilation des termes formés au moyen d'opérateurs génériques, notre propos étant de montrer comment passer de ces termes génériques à des termes fonctionnels de deuxième ordre.

Nous supposons connu du lecteur le langage Lisp, et ne rappellerons que quelques détails relatifs à la norme Common Lisp [170], choisie pour l'implantation de FP2. Pour donner rapidement quelques précisions techniques au sujet de cette dernière[37], le compilateur-interpréteur FP2, y compris le traite-

[34] Notons que [95], qui est une formalisation de la compilation des systèmes de réécriture conditionnels, se donne comme objectif final la compilation d'OBJ.

[35] Les processus sont eux aussi compilés.

[36] ... et aussi des raisons d'homogénéisation dans la notation de tous les codes générés. Nous donnerons au chapitre 4 la compilation des termes avec exception : le code généré utilise des formes spéciales de Common Lisp qui ne sont pas issues du λ-calcul.

[37] Réalisée en collaboration avec Philippe Schnoebelen.

ment des processus, comporte entre 11 000 et 12 000 lignes de texte source, il est adapté aussi bien à KCL (Kyoto Common Lisp) [179] — sur SUN et sur VAX — qu'à Lucid (Sun Common Lisp) [183].

Alors que le choix d'une machine abstraite comporte le risque de mal présumer des ressources d'une architecture réelle ayant à supporter le code, la compilation en Lisp permet de ne pas se soucier d'utiliser la machine cible au mieux de ses possibilités. Sur ce point précis, c'est à la fois un avantage et un inconvénient, car tout repose sur le compilateur Lisp dont on dispose.

De façon générale, l'inconvénient majeur de la génération de fonctions Lisp est lié au fait que nous compilons un langage fortement typé en un langage dans lequel tous les tests de sorte sont effectués dynamiquement. Par exemple, lors de l'évaluation de l'expression `(+ x y)`, Lisp vérifie dynamiquement que `x` et `y` sont bien des nombres, alors qu'à l'exécution, une erreur de sorte est impossible[38] dans une expression Lisp provenant de la compilation d'un terme FP2. Remarquons cependant que les fonctions Lisp générées par le compilateur FP2 peuvent à leur tour être compilées, et qu'il est possible, en Common Lisp, d'indiquer au compilateur de supprimer certains tests dynamiques.

Outre les avantages de toute compilation — rapidité et sûreté d'exécution, possibilité d'appliquer en profondeur des techniques d'optimisation du code —, la compilation des opérateurs de FP2 en fonctions Lisp permet d'obtenir un code relativement lisible et d'étendre facilement à FP2 un certain nombre d'outils pour la mise au point fournis avec Lisp : trace de fonctions, exécution pas à pas ou avec des points d'arrêt, pisteur...

Le "cahier des charges" du compilateur de la partie fonctionnelle du langage FP2 est le suivant :

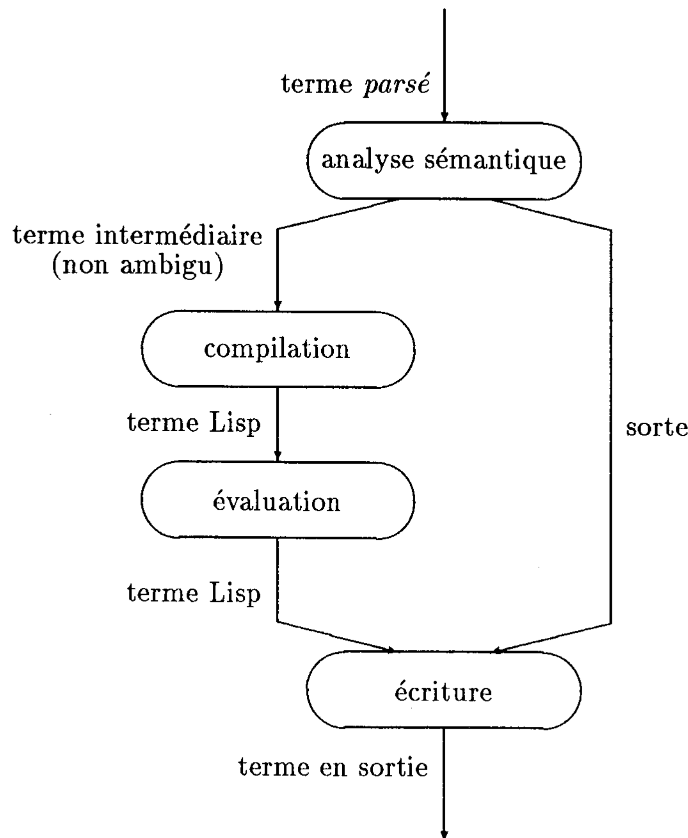
- choisir une représentation pour chaque sorte et compiler en conséquence les constructeurs ;
- compiler chaque opérateur non constructeur d'après ses règles de réécriture.

2.1 Principes généraux

2.1.1 Boucle d'évaluation de FP2

Le schéma suivant retrace les diverses étapes de l'évaluation d'une expression FP2 :

[38]Sauf s'il reste des *bugs*, bien entendu !



Le résultat de l'analyse sémantique est un terme intermédiaire dans lequel les ambiguïtés dues à la surcharge ont été levées et les modèles des opérateurs complètement explicités. Les règles utilisées pour l'analyse sémantique sont données au chapitre 3. Les homonymies, dues à la surcharge ou à une éventuelle coïncidence entre le nom d'un opérateur à compiler et le nom d'une fonction Common Lisp qui existe déjà, imposent de choisir pour chaque opérateur un nom interne. Étant donné que ce problème est traité en amont de la compilation proprement dite, nous ne le considérons pas ici pour des raisons de simplicité, et, suivant la convention entrevue au chapitre 0, nous utilisons le même nom, éventuellement assorti d'un indice ("$=_N$", "$=_S$"), pour un opérateur FP2 et la fonction Lisp correspondante.

Le côté "fortement typé" de FP2 permet de ne garder aucune information de sorte durant l'évaluation d'un terme compilé, et permet en outre à plusieurs sortes de pouvoir partager la même représentation. Il devient alors nécessaire de fournir la sorte de l'expression, obtenue par l'analyse sémantique, à la fonction chargée de retrouver la représentation externe que connaît l'utilisateur.

L'idée directrice de cette décomposition en termes *parsés*, termes intermédiaires et termes compilés est de pouvoir *ignorer* la représentation des termes compilés lors des calculs statiques, qui sont tous effectués sur les termes intermédiaires. Aussi, les termes compilés ne sont utilisés que lors de l'évaluation d'une expression ou la simulation d'un processus. Les calculs statiques regroupent les vérifications sémantiques et, en sus pour la partie parallélisme, le calcul des règles des processus spécifiés au moyen des opérateurs vus au §0.2.2. Les principes de ce calcul peuvent être trouvés en [160] et une spécification en [162]. À noter toutefois que l'implantation est plus performante.

2.1.2 Compilation des sortes

2.1.2.1 Approche générale

De façon générale, on appelle **implantation** la réalisation, à partir de fonctionnalités qui existent déjà, de nouvelles fonctionnalités de plus haut niveau. Cette notion regroupe un large éventail d'applications, selon les diverses fonctionnalités considérées, et selon les procédés employés. Notre réalisation rentre effectivement dans ce cadre puisqu'il s'agit de l'implantation d'un langage (ici, FP2) au moyen d'un autre langage (ici, Common Lisp). Dans le domaine de la spécification algébrique, une approche intéressante de l'implantation est sa formalisation à l'aide des outils et des méthodes de cette discipline : on parle alors d'**implantation abstraite**. L'avantage de cette démarche est de pouvoir établir des critères formels de correction d'une implantation. Cette approche a été étudiée de façon très détaillée dans [10]. Nous en rappelons ci-après les bases, avant d'aborder plus précisément le *modus operandi* retenu pour FP2.

L'implantation abstraite revient donc à considérer que l'on dispose de sortes $s_1, \ldots, s_n$, et que l'on cherche à les utiliser pour implanter une nouvelle sorte s. Bien sûr, les sortes déjà implantées $s_1, \ldots, s_n$ sont spécifiées algébriquement, ainsi que la nouvelle sorte s. Deux méthodes viennent à l'esprit :

Abstraction Suivant cette approche, on s'attache à obtenir les termes de sorte s à partir des termes de sortes $s_1, \ldots, s_n$. L'opérateur qui réalise cette *abstraction* a donc le profil :

$$s_1 \times \cdots \times s_n \rightarrow s$$

Il est alors nécessaire que cette opérateur soit interprété par une fonction surjective. De plus, rien n'assure que cette fonction ne synthétise pas d'éléments *en sus* de ceux qui représentent les objets à implanter, comme rien n'assure qu'elle soit injective. Il est donc indispensable de :

- caractériser l'image réciproque du type s, autrement dit dégager les objets de $s_1 \times \cdots \times s_n$ qui sont vraiment des implantations des objets de sorte s : c'est la donnée des *invariants de représentation*,
- caractériser les éléments de $s_1 \times \cdots \times s_n$ qui représentent le même terme donné quelconque de sorte s : c'est la donnée de la *représentation de l'égalité*,

et l'inconvénient de cette approche apparaît dès lors : on ne connaît aucune méthode générale permettant de déduire les invariants de représentation et la représentation de l'égalité à partir de la spécification.

Représentation C'est l'approche symétrique qui consiste à associer à chaque terme de sorte s un n-uplet de sorte $s_1 \times \cdots \times s_n$. Elle résout de par son principe le problème des invariants de représentation, mais soulève des difficultés au niveau de son interprétation algébrique, dès lors que l'on considère un *opérateur* de restriction. En effet, si l'on souhaite formaliser la représentation du type s dans le cadre algébrique, et exprimer les équations qui la définissent, le codomaine de cet opérateur doit être une sorte, et non un produit de sortes. On en vient ainsi à créer un opérateur de profil $s_1 \times \cdots \times s_n \rightarrow s$, qui, en fait, n'est autre que l'opérateur d'abstraction précédent. Dès lors, l'opérateur de représentation a le profil $s \rightarrow s$, et représente la fonction d'identité.

Pour pallier ces difficultés et insuffisances, Gilles Bernot a proposé dans [10] une approche qui combine abstraction et représentation, ou plus exactement *synthèse* et représentation. Il définit une

implantation abstraite comme étant la donnée d'un quintuplet :

$$\langle \rho, \Sigma_{SYNTH}, C, A_{OP}, A_{EQ} \rangle$$

où :

- ρ est la représentation : c'est un isomorphisme de signatures reliant la signature que l'on implante à la signature dite *constructive* — pour une sorte s à implanter, on note $\bar{s}$ la sorte $\rho(s)$ qui la représente, et de même, pour tout opérateur ω à implanter, on note $\bar{\omega} = \rho(\omega)$;
- Σ_{SYNTH} est l'ensemble des opérateurs de synthèse : pour chaque sorte constructive $\bar{s}$, Σ_{SYNTH} contient un unique opérateur :

$$\langle \cdot, \ldots, \cdot \rangle : s_1 \times \cdots \times s_n \rightarrow \bar{s}$$

  $s_1, \ldots, s_n$ étant des sortes déjà implantées ;
- C regroupe la spécification des sortes et opérateurs cachés ;
- A_{OP} est l'ensemble d'axiomes[39] qui décrit l'implantation des opérateurs constructifs ;
- A_{EQ} est un ensemble d'axiomes qui définit la représentation de l'égalité.

Les opérateurs de Σ_{SYNTH} sont en fait des constructeurs de sortes produits : pour toute sorte s, l'opérateur $\langle \cdot, \ldots, \cdot \rangle : s_1 \times \cdots \times s_n \rightarrow \bar{s}$ synthétise les n-uplets de sortes $s_1, \ldots, s_n$ en des termes de sorte $\bar{s}$ que l'on restreint ensuite en élaguant les termes qui ne représentent aucun objet de sorte s, d'où la notion de synthèse plutôt que d'abstraction. Il est par ailleurs important de mentionner que si ce formalisme est quelque peu lourd, des conventions très simples permettent de le rendre totalement transparent à l'utilisateur : ce dernier peut alors laisser implicites les sortes constructives et l'isomorphisme de représentation (voir les exemples donnés dans [10]).

Les critères de correction d'une telle implantation sont :

- tout terme fermé à implanter doit posséder une représentation,
- l'algèbre qui est la sémantique de l'implantation doit être isomorphe à l'algèbre de la présentation considérée au départ.

Outre une formalisation rigoureuse et complète de l'implantation abstraite, le principal apanage de ce formalisme est que les preuves de correction se ramènent à la résolution de problèmes bien connus en spécification algébrique, que l'on sait traiter de manière formelle. Ce n'est pas le cas pour l'approche par abstraction où les critères de correction ne s'expriment que d'une manière purement sémantique, leur traitement nécessitant alors une description des algèbres modèles de l'implantation.

2.1.2.2 Approche de FP2

Il serait tout à fait envisageable de mener une étude complètement formelle du procédé de compilation retenu pour la partie fonctionnelle de FP2. Elle commencerait par une description des structures de données utilisées dans la langage fonctionnel cible (en l'occurrence, Common Lisp), suivie d'une formulation des procédures de calcul de ce langage. On donnerait ensuite les règles de compilation, et

[39] Dans [10], Gilles Bernot appelle "axiome" une équation conditionnelle au sens de [120].

il resterait à prouver que l'algèbre constituée par les formes compilées est isomorphe à l'algèbre des termes de la présentation algébrique de départ. [164] traite de la compilation du filtrage suivant une telle approche.

Dans ce qui va suivre, nous n'allons pas nous situer à un niveau aussi formel, mais seulement nous attacher à faire ressortir les principes de compilation retenus. Nous allons tenter d'en donner une vue relativement informelle sans toutefois trop renoncer à la précision. Comme nous l'avons annoncé en début de ce chapitre, c'est surtout sur la compilation des opérateurs génériques que nous insisterons (cf. §2.2).

Étant donné une sorte, les manipulations qu'à l'exécution, on souhaite pouvoir effectuer sur ses termes sont liées au filtrage des parties gauches des règles de réécriture, le résultat de la compilation d'une partie gauche étant une conjonction de conditions correspondant à ce filtrage. Il est par conséquent nécessaire de savoir si un terme est construit à l'aide d'un constructeur particulier, et dans l'affirmative, de pouvoir récupérer tel ou tel argument de ce constructeur.

En conséquence, dans le but de rendre la représentation totalement transparente, nous définissons, en même temps que chaque constructeur, son **discriminant**[40] et ses **sélecteurs**. Ainsi, pour un constructeur g d'arité n, la représentation la plus "immédiate", celle qui vient en premier à l'esprit, est :

```
(defun g (x1 ... xn)
  (list 'g x1 ... xn))
```

c'est-à-dire le nom du constructeur, considéré ici comme une étiquette, suivi de ses arguments. Si nous admettons que tous les éléments de cette sorte sont représentés par des listes, le discriminant correspondant est :

```
(defun is-g (x)
  (eq (car x) 'g))
```

et les sélecteurs sont :

```
(defun get-g-i (x)
  (nth i x))
```

$(1 \leqslant i \leqslant n)$

Lors du traitement d'un opérateur non constructeur f, il est alors immédiat de compiler une règle de réécriture telle que :

$$f(g(x_1, \ldots, x_n)) \longrightarrow f_0(x_1, \ldots, x_n)$$

en la clause Lisp :

```
((is-g x) (f0 (get-g-1 x) ... (get-g-n x)))
```

Cette méthode intègre parfaitement les cas où l'on connaît une représentation plus adaptée à une sorte particulière. Ainsi, pour représenter les termes de sorte *Nat* par les entiers naturels de Common Lisp, il suffit de générer les fonctions suivantes :

```
(defun zero ()
  0)

(defun is-zero (x)
  (zerop x))

(defun succ (x)
  (1+ x))

(defun is-succ (x)
  (plusp x))

(defun get-succ-1 (x)
  (1- x))
```

[40] En anglais : *testor*.

Poursuivant dans cette voie, les entiers naturels de Common Lisp peuvent être utilisés pour représenter non seulement les termes de sorte *Nat*, mais aussi les termes de toute sorte s définie au moyen de deux constructeurs c_1 et c_2 dont les profils sont :

$$\begin{array}{lcccc} c_1 & : & - & \rightarrow & s \\ c_2 & : & s & \rightarrow & s \end{array}$$

c'est-à-dire toute sorte dont les constructeurs forment une signature isomorphe — au sens d'un morphisme de signatures dans la catégorie **Sig** — à celle des constructeurs de la sorte *Nat*.

Plus généralement, FP2 prévoit une représentation par défaut sous forme d'atomes pour les constructeurs constants, sous forme de listes pour les autres, et il possède une "base" de sortes, pour lesquelles il connaît et réalise une compilation plus performante. Les constructeurs de cette base de sortes sont donnés à un isomorphisme de signatures près.

La compilation des sortes de FP2 est résumée dans le tableau de la p. 86. On remarquera que les cas particuliers de représentation se subdivisent en les quatre possibilités suivantes :

(1) tous les constructeurs sont constants ;

(2) la signature des constructeurs est isomorphe à celle des constructeurs de la sorte *Nat* ;

(3) un seul constructeur est d'arité supérieure ou égale à 2, et les autres constructeurs sont constants ;

(4) la sorte n'admet qu'un seul constructeur.

Les sélecteurs sont immédiats à expliciter. Nous avons donné les discriminants afin de souligner qu'ils effectuent le minimum de tests : par exemple, dans le cas (3-a), le discriminant `is-c`$_1$ est équivalent à la fonction **endp**[41] car les termes de cette sorte n'admettent pas d'autre atome que `nil`, alors que dans le cas (3-b), `is-c`$_1$ est donné équivalent à la fonction `null`, à cause du choix de représentation sous forme d'atomes pour les constructeurs $c_2, \ldots, c_{p-1}$.

On aura remarqué que les séquences de FP2 (cf. exemple 0.87) et les arbres génériques quelconques — la spécification du type *Tree* est donnée à l'annexe (cf. exemple B.13 et §B.3) — sont un cas particulier du cas (3-a), et sont par conséquent représentés par des listes Lisp. De même, la sorte *Bool* est un cas particulier du cas (1-a). Enfin, nous admettrons que, conformément à l'utilisation habituelle des valeurs logiques en Lisp, l'opérateur *false* est compilé en `nil`, et l'opérateur *true* en `t`.

Note En fait, la compilation des constructeurs n'est pas aussi immédiate si l'opérateur est partiel. Ce point, qui sera abordé au chapitre 4, n'interfère pas avec le choix d'une représentation pour la sorte, qui s'effectue uniquement d'après les profils de ses constructeurs.

2.1.2.3 Propriétés de correction

Nous allons à présent formaliser davantage les notions que nous venons de dégager. Pour un terme t de sorte s, nous notons $cp(t)$ la forme compilée de ce terme. Considérons l'image, par l'opération de compilation, de tous les termes de sorte s. Il est essentiel que l'application du discriminant associé à un constructeur g de la sorte s, renvoie la valeur "vrai" si et seulement s'il est appliqué à l'image

[41] La fonction **endp** ne retourne un résultat que lorsque son argument est une liste. Son implantation peut par conséquent être plus performante que celle de la fonction `null`, qui s'applique à toute expression Common Lisp (cf. [170]).

CONSTRUCTEURS	FORMES COMPILÉES	DISCRIMINANTS
(1-*a*) $c_1 : \quad - \quad \rightarrow s$ $c_2 : \quad - \quad \rightarrow s$	(defun c_1 () nil) (defun c_2 () t)	(defun is-c_1 (x) (null x)) (defun is-c_2 (x) x)
(1-*b*) $c_1 : \quad - \quad \rightarrow s$ $c_2 : \quad - \quad \rightarrow s$ $\vdots$ $c_p : \quad - \quad \rightarrow s$ $p > 2$	(defun c_1 () nil) (defun c_2 () 'c_2) $\vdots$ (defun c_p () 'c_p)	(defun is-c_1 (x) (null x)) (defun is-c_2 (x) (eq x 'c_2)) $\vdots$ (defun is-c_p (x) (eq x 'c_p))
(2) $c_1 : \quad - \quad \rightarrow s$ $c_2 : \quad s \quad \rightarrow s$	(defun c_1 () 0) (defun c_2 (x_1) (1+ x_1))	(defun is-c_1 (x) (zerop x)) (defun is-c_2 (x) (plusp x))
(3-*a*) $c_1 : \quad - \quad \rightarrow s$ $c_2 : s_1 \times \ldots \times s_n \rightarrow s$ $n \geq 2$	(defun c_1 () nil) (defun c_2 (x_1 ... x_n) (list* x_1 ... x_n))	(defun is-c_1 (x) (endp x)) (defun is-c_2 (x) x)
(3-*b*) $c_1 : \quad - \quad \rightarrow s$ $c_2 : \quad - \quad \rightarrow s$ $\vdots$ $c_{p-1} : \quad - \quad \rightarrow s$ $c_p : s_1 \times \ldots \times s_n \rightarrow s$ $n \geq 2, p > 2$	(defun c_1 () nil) (defun c_2 () 'c_2) $\vdots$ (defun c_{p-1} () 'c_{p-1}) (defun c_p (x_1 ... x_n) (list* x_1 ... x_n))	(defun is-c_1 (x) (null x)) (defun is-c_2 (x) (eq x 'c_2)) $\vdots$ (defun is-c_{p-1} (x) (eq x 'c_{p-1})) (defun is-c_p (x) (consp x))
(4-*a*) $c_1 : \quad s_1 \quad \rightarrow s$	(defun c_1 (x_1) x_1)	(defun is-c_1 (x) t)
(4-*b*) $c_1 : s_1 \times \ldots \times s_n \rightarrow s$	(defun c_1 (x_1 ... x_n) (list* x_1 ... x_n))	(defun is-c_1 (x) t)
AUTRES CAS $c_i : \quad - \quad \rightarrow s$ $c_j : s_1 \times \ldots \times s_n \rightarrow s$ pour c_i, c_j constructeurs quelconques	(defun c_i () 'c_i) (defun c_j (x_1 ... x_n) (list* 'c_j x_1 ... x_n))	(defun is-c_i (x) (eq x 'c_i)) (defun is-c_j (x) (and (consp x) (eq (car x) 'c_j)))

d'un terme dont la racine est g. Autrement dit, il est essentiel que les discriminants satisfassent les propriétés de correction suivantes :

$$\left.\begin{array}{lll} t \equiv g' & \Longleftrightarrow & (\texttt{is-g'}\ cp(t)) \text{ se réduit à "vrai"} \\ t \equiv g''(u_1,\ldots,u_n) & \Longleftrightarrow & (\texttt{is-g''}\ cp(t)) \quad \ldots \end{array}\right\} \qquad (2.1)$$

pour tous opérateurs constructeurs g' d'arité 0 et g'' d'arité n ($n \geqslant 0$), et tous termes bien formés t, $(u_i)_{1 \leqslant i \leqslant n}$, exprimés uniquement à l'aide de constructeurs.

De même, on exigera que les sélecteurs d'un constructeur g'' d'arité non nulle soient tels que :

$$t \equiv g''(u_1,\ldots,u_n) \Longrightarrow \texttt{get-g''-}i(cp(t)) = cp(u_i) \qquad (1 \leqslant i \leqslant n) \qquad (2.2)$$

Proposition 2.1 *La compilation des opérateurs constructeurs de FP2 vérifie les propriétés (2.1) et (2.2).*

En ce qui concerne la propriété (2.1), qui caractérise les discriminants, ceci est facile à vérifier pour tous les cas de représentation du tableau qui figure p. 86. Bien sûr, si l'un des avantages de cette méthode est de pouvoir éventuellement ajouter à la "base" de sortes d'autres cas de compilations performantes, il est néanmoins essentiel de s'assurer que ces nouveaux choix de compilation satisfont les propriétés (2.1) et (2.2).

2.1.3 Compilation des opérateurs

La compilation d'un opérateur vise à transformer des règles de réécriture en une fonction Lisp. FP2 adopte la convention selon laquelle les règles de réécriture d'un opérateur sont examinées séquentiellement, suivant l'ordre dans lequel elles ont été écrites[42]. Le cas des opérateurs non complètement définis par rapport aux constructeurs sera abordé au chapitre 4. Remarquons dès à présent que d'après la convention sur l'ordre des règles de réécriture, il suffit, pour tenir compte de cette éventualité, d'ajouter en queue des règles, pour chaque opérateur, une règle supplémentaire "trappe" dont la partie gauche est la plus générale possible (c'est-à-dire un terme linéaire dont tous les sous-termes à la profondeur 1 sont des variables), et la partie droite le traitement des cas de non-définition de l'opérateur.

Toute règle de réécriture peut être compilée en une clause Lisp, dont le premier élément est une condition équivalente au filtrage de la partie gauche, et le second la compilation de la partie droite. Nous n'exposons pas cette dernière en détail, étant donné que les constructions usuelles utilisées dans une partie droite (*if, let*) ont un équivalent en Common Lisp (formes spéciales `if`, `let*`), et traiterons en détail uniquement la compilation des termes génériques au §2.2. Quant à la compilation d'une partie gauche, elle devient immédiate dès lors que pour chaque constructeur, son discriminant et ses sélecteurs sont connus. Ainsi, la compilation la plus immédiate d'un opérateur tel que :

$$\begin{array}{rcl} fact(0) & \texttt{==>} & 1 \\ fact(succ(n)) & \texttt{==>} & succ(n) * fact(n) \end{array} \qquad (2.3)$$

est :

[42] D'un point de vue théorique, ce comportement se formalise au moyen de systèmes de réécriture *avec priorités* [5]. Dans notre cas, la priorité des règles décroît strictement suivant l'ordre d'apparition.

```
(defun fact (x)
  (cond ((is-0 x) 1)
        ((is-succ x) (* (succ (get-succ-1 x)) (fact (get-succ-1 x))))
        (t (nothing-matches))))
```

(où "(nothing-matches)" est le traitement prévu lorsque l'opérateur n'est pas complètement défini par rapport aux constructeurs).

Un code plus performant est obtenu par recherche des sous-expressions communes :

```
(defun fact (x)
  (cond ((is-0 x) 1)
        ((is-succ x) (* x (fact (get-succ-1 x))))
        (t (nothing-matches))))
```

en "factorisant" l'expression $succ(n)$ qui apparaît à deux reprises dans la règle (2.3). On peut voir également que la meilleure compilation est :

```
(defun fact (x)
  (if (is-0 x) 1 (* x (fact (get-succ-1 x)))))
```

En effet, si l'on examine la seconde règle, c'est que l'évaluation de l'expression `(is-0 x)` a retourné la valeur "faux". Les seuls constructeurs de *Nat* étant 0 et *succ*, si `x` est de sorte *Nat*, `(is-succ x)` se réduit nécessairement à "vrai". Par suite, non seulement il est inutile d'évaluer `(is-succ x)`, mais en outre, le domaine de *fact* étant complètement couvert, on peut éliminer la règle "trappe".

Cette meilleure compilation du filtrage, que nous ne détaillons pas davantage ici, est décrite et prouvée dans [164]. Elle est obtenue en compilant chaque règle avec un argument de plus représentant la portion du domaine de l'opérateur qu'il reste à couvrir. Nous terminons ce bref survol de la compilation des opérateurs en signalant [95], qui part d'une approche plus générale que [164], et [121], qui étudie des améliorations du code obtenues par factorisation des sous-expressions communes et évaluation paresseuse de ces sous-expressions.

2.1.4 "Décompilation" — Traitement des formats externes spéciaux

Outre leur utilisation pour l'évaluation des conditions correspondant aux filtrages, les discriminants et les sélecteurs servent à la "décompilation", c'est-à-dire à la conversion d'un terme compilé en un terme intermédiaire. La fonction d'écriture doit réaliser cette opération, suivie du remplacement du nom interne de chaque opérateur par son nom externe[43]. La "décompilation" consiste à appliquer jusqu'au succès les discriminants correspondant aux constructeurs, puis à appeler récursivement cette opération sur le résultat de chaque sélecteur. Soit x le résultat de la compilation d'un terme quelconque de sorte s :

decompile(x, s) **is**
for g $\in$ *constructors*(s) **do**
    if is-g(x)
        then let n :: arity(g)

[43] À noter que cette dernière opération s'effectue de manière totalement univoque, contrairement à la détermination du nom interne d'un opérateur, la surcharge pouvant être source d'ambiguïtés (cf. chapitre 3).

in if $n = 0$ **then exit**($\overline{\text{g}}$)
  else let $s_1 \times \cdots \times s_n$ :: dom(g)
    in exit($\overline{\text{g}(decompile(\texttt{get-g-1}(x), s_1), \ldots, decompile(\texttt{get-g-}n(x), s_n))}$)
   endlet
  endif
 endlet
endif
endfor

Théorème 2.2 *Pour tout terme t de sorte s, exprimé uniquement à l'aide de constructeurs :*

$$decompile(cp(t), s) \equiv t$$

Preuve Elle est aisée, par récurrence sur la profondeur du terme t. C'est une conséquence de la proposition 2.1, l'absence d'équations entre constructeurs assurant l'égalité syntaxique.

Un autre avantage de cette "décompilation" est l'implantation efficace d'une facilité importante pour la convivialité du langage : le traitement des formats externes spéciaux. S'il est souhaitable, par exemple, que les entiers puissent être spécifiés sous leur forme décimale par l'utilisateur, il paraît évident que le bénéfice de cette notation "abrégée" est perdu si la fonction de *parsing* développe un entier non nul quelconque i dans sa représentation en base 1 (c'est-à-dire $succ^i(0)$). On peut remarquer que les entiers sous forme décimale coïncident avec la forme compilée des objets de sorte *Nat*. Aussi, nous introduisons une nouvelle catégorie de termes, appelée "objets-Lisp", caractérisés par la donnée d'un terme compilé et de sa sorte. Ceci suppose que la représentation de cette sorte et le résultat du *parsing* d'un tel terme sont compatibles, et permet la compilation immédiate d'un opérateur f tel que :

$$\begin{aligned} f(1989) &==> true \\ f(n) &==> false \end{aligned} \quad (2.4)$$

en :

```
(defun f (x)
  (if (= x 1989) (true) (false)))
```

sans nécessité de développer la règle (2.4) en :

$$f(succ^{1989}(0)) ==> true$$

Plus généralement, cette notion est fort utile pour les traitements effectués sur les termes intermédiaires, car on peut ainsi ne destructurer ces objets suivant les constructeurs qu'en cas de besoin uniquement. Ainsi, si dans la partie parallélisme, le traitement d'un opérateur de connexion entraîne l'unification des termes $succ(x)$ (x de sorte *Nat*) et 1989, on peut traiter statiquement cette unification. Puisque l'expression (`is-succ` 1989) se réduit à "vrai", elle est équivalente à l'unification de x et (`get-succ-1` 1989), d'où l'on déduit la solution $[x \mapsto 1988]$.

Cette méthode permet d'intégrer à une démarche générale les formats externes spéciaux (entiers, caractères, chaînes de caractères), sans effectuer de destructuration inutile, et sans traiter à part chaque format spécial.

Note Nous n'avons pas réalisé d'interface avec Common Lisp, qui permette d'utiliser directement des fonctions Lisp dans une spécification FP2. Dans le langage OBJ3, une telle interface est réalisée au moyen des ***built-in equations***.

```
obj NAT-SQRT is
   protecting NAT .
   op  sqrt- : Nat -> Nat .  ---  ⌊√ ⌋
   op  sqrt+ : Nat -> Nat .  ---  ⌈√ ⌉
   var x     : Nat .
   bq  sqrt-(x) = (values (floor (sqrt x))) .
   bq  sqrt+(x) = (values (ceiling (sqrt x))) .
jbo
```

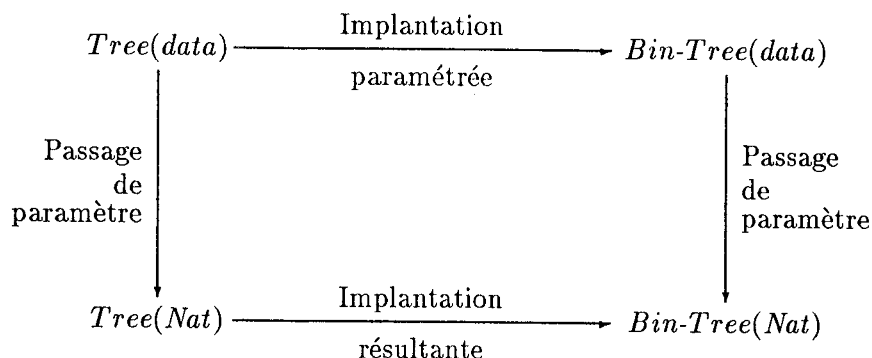
Signalons, pour clore l'exposé de ces principes généraux, que le compilateur peut utiliser à des fins d'optimisation les définitions des constructeurs, discriminants et sélecteurs. Ainsi, en utilisant que *false* et *true* sont représentés respectivement par `nil` et `t`, le compilateur effectue les optimisations suivantes lors de la compilation de l'opérateur f précédent :

```
(if (= x 1989) (true) (false))  ⟶  (if (= x 1989) t nil)
                                ⟶  (= x 1989)
```

2.2 Compilation des termes génériques

2.2.1 Un résultat théorique

Dans [10], Gilles Bernot ne traite pas l'implantation abstraite de présentations génériques, mais indique que son formalisme devrait sans difficulté s'étendre à ce cas. Sous une forme fonctorielle dont on peut retrouver certains aspects en [10], les auteurs de [57] étudient l'implantation d'une présentation paramétrée $(\mathcal{P}_0, \mathcal{P})$ par une présentation paramétrée $(\mathcal{P}_0, \mathcal{P}')$. Un exemple de ce procédé est l'implantation des arbres quelconques — *Tree(data)*, où *data* désigne une sorte formelle — par des arbres binaires — *Bin-Tree(data)*. Un résultat important est que si l'implantation de la présentation paramétrée $(\mathcal{P}_0, \mathcal{P})$ par la présentation paramétrée $(\mathcal{P}_0, \mathcal{P}')$ est correcte, alors toute instanciation de la présentation $\mathcal{P}$ est correctement implantée par la même instanciation appliquée à la présentation $\mathcal{P}'$. Reprenant l'exemple, on peut traduire ce dernier résultat comme suit dans le cas d'un remplacement $[data \mapsto Nat]$:



Cette propriété, vraie pour toute présentation paramétrée persistante, est résumée dans [57] par : "un passage de paramètre correct commute avec une implantation correcte d'une présentation paramétrée".

2.2.2 Règles

Le principe que nous avons adopté pour la compilation des termes génériques a été esquissé pour la première fois dans [162], de manière informelle. Il consiste à prévoir pour les fonctions compilées un argument supplémentaire qui représente les opérateurs du modèle et à exprimer ces derniers sous forme d'expressions de fonctions avec fermeture lexicale[44].

FP2 étant un langage fortement typé, aucune information de sorte n'est nécessaire pendant une exécution. Nous ne conservons donc pas les sortes d'une expression représentant un modèle. En revanche, lorsque la propriété exigée d'un opérateur générique f contient des opérateurs dans sa signature, nous donnons un argument supplémentaire à la fonction Common Lisp qui représente f (dans le cas contraire, le nombre d'arguments de la fonction Lisp est égal à l'arité de l'opérateur FP2). Cet argument supplémentaire — par convention, nous le désignerons dans toute la suite par l'identificateur `fm` — représente donc la forme compilée des opérateurs présents dans l'expression de modèle qui qualifie l'opérateur générique.

Pour la partie gauche d'une règle de réécriture quelconque, les sous-termes de sorte formelle licites sont uniquement les variables, les constructeurs d'une sorte formelle étant *a priori* inconnus. En conséquence, l'algorithme décrit dans [164] est tout à fait utilisable dans le cadre des opérateurs génériques, une variable de sorte formelle représentant une couverture complète du type correspondant. Nous allons donc nous borner à exposer le traitement des parties droites, composées de termes génériques, donnés avec leur modèle. À noter que c'est ce même traitement qui est utilisé par l'interpréteur pour compiler les termes dont on demande l'évaluation, le modèle exigé étant dans ce dernier cas le modèle trivial "$\diamond$".

Détaillons à présent les conventions et notations que nous allons employer, y compris en ce qui concerne le choix des styles de caractères, destiné à différencier, d'une part, l'ossature des expressions Common Lisp générées par le compilateur FP2, d'autre part, les appels des fonctions du compilateur lui-même. Comme on a pu le remarquer dans les exemples précédents, nous avons utilisé le style "`machine à écrire`" pour désigner des expressions Lisp, ne conservant le style "*italique mathématique*" que pour les paramètres ou les résultats générés par nos fonctions de compilation. Les notations utilisées dans les deux tableaux suivants sont :

[44]En Common Lisp, une fermeture *lexicale* est réalisée au moyen de la forme spéciale `function`, c'est-à-dire que la portée des variables liées s'applique à l'intérieur de la forme `function` :

```
(defun add-curry (x)
  (function (lambda (y) (+ x y))))
```

`(add-curry 2)` est l'expression de fonction : `(function (lambda (y) (+ 2 y)))`

d'où :

```
(funcall (add-curry 2) 3)
= 5
```

(Rappelons que l'expression "`#'...`" — que nous emploierons plus loin — est une abréviation pour "`(function ...)`".)
Si le dialecte Lisp considéré est à liaison dynamique, alors dans l'application d'une fonction telle que :

```
(defun add-curry (x)
  (lambda (y) (+ x y)))
```

la variable `x` de l'expression `(+ x y)` a une portée locale à la forme `lambda` et devra être liée dynamiquement à chaque appel de la fonction `add-curry`.

ϕ	désigne un opérateur formel quelconque ;
f	désigne un opérateur générique quelconque ;
`f`	désigne le nom interne de l'opérateur générique f précédent ;
RM	désigne le modèle exigé du module (type ou enrichissement) en cours de compilation ;
$position_{RM}(\phi)$	donne le rang de l'opérateur formel ϕ dans la liste des opérateurs formels du modèle exigé RM ;
M	désigne un modèle quelconque (qui peut éventuellement coïncider avec RM) ;
$operators\text{-}in(\diamond)$	$= false$;
$operators\text{-}in(p[t_1,\ldots,t_u\ /\ ])$	$= false$;
$operators\text{-}in(p[t_1,\ldots,t_u\ /\ \tilde{f}_1,\ldots,\tilde{f}_v])$	$= true$;
ξ	désigne une variable quelconque.

Dans l'expression "$p[t_1,\ldots,t_u\ /\ \tilde{f}_1,\ldots,\tilde{f}_v]$", tout $\tilde{f}_i$, pour $1 \leqslant i \leqslant v$, représente soit un opérateur formel, soit un opérateur générique donné avec son modèle (cf. §1.2.7). Rappelons également que selon les conventions de Lisp, les positions sont comptées à partir de 0, c'est-à-dire que pour tout i, $1 \leqslant i \leqslant v$:

$$position_{p[t_1,\ldots,t_u\ /\ \phi_1,\ldots,\phi_v]}(\phi_i) = i - 1$$

Voici à présent la définition complète de la fonction cp_{RM}, qui compile un terme générique quelconque présent dans la partie droite d'une règle de réécriture d'un module dont le modèle exigé est RM. Le résultat est, bien sûr, une expression Common Lisp.

$$cp_{RM}(\phi(w_1,\ldots,w_n)) = (\texttt{funcall (nth } position_{RM}(\phi)\texttt{ fm)}\ cp_{RM}(w_1)\ \ldots\ cp_{RM}(w_n)\texttt{)} \tag{2.5}$$

$$cp_{RM}(f.M(w_1,\ldots,w_n)) = (\texttt{f } cp_{RM}(w_1)\ \ldots\ cp_{RM}(w_n)\texttt{)}\ \ \textbf{if}\ \neg operators\text{-}in(M) \tag{2.6}$$

$$cp_{RM}(f.RM(w_1,\ldots,w_n)) = (\texttt{f } cp_{RM}(w_1)\ \ldots\ cp_{RM}(w_n)\texttt{ fm)}\ \ \textbf{if}\ operators\text{-}in(RM) \tag{2.7}$$

$$cp_{RM}(f.p[t_1,\ldots,t_u\ /\ \tilde{f}_1,\ldots,\tilde{f}_v](w_1,\ldots,w_n)) = (\texttt{f } cp_{RM}(w_1)\ \ldots\ cp_{RM}(w_n)\ (\texttt{list } ge_{RM}(\tilde{f}_1)\ \ldots\ ge_{RM}(\tilde{f}_v)\texttt{))} \tag{2.8}$$

$ge_{RM}(\tilde{f}_i)$ retourne la forme compilée de $\tilde{f}_i$, employée à l'intérieur de l'argument qui représente le "modèle compilé" de l'opérateur générique en cours de traitement.

$$ge_{RM}(\phi) = \texttt{(nth } position_{RM}(\phi) \texttt{ fm)} \quad (2.9)$$

$$ge_{RM}(f.M) = \texttt{\#'f} \quad \text{if } \neg operators\text{-}in(M) \quad (2.10)$$

$$ge_{RM}(f.RM) = \texttt{\#'(lambda (x}_1 \ldots \texttt{x}_n\texttt{) (f x}_1 \ldots \texttt{x}_n \texttt{ fm))} \quad \text{where } n = \text{arity}(f) \quad \text{if } operators\text{-}in(RM) \quad (2.11)$$

$$ge_{RM}(f.p[t_1,\ldots,t_u \;/\; \tilde{f}_1,\ldots,\tilde{f}_v]) = \texttt{\#'(lambda (x}_1 \ldots \texttt{x}_n\texttt{)} \; \texttt{(f x}_1 \ldots \texttt{x}_n \texttt{ (list } ge_{RM}(\tilde{f}_1) \ldots ge_{RM}(\tilde{f}_v)\texttt{)))} \quad \text{where } n = \text{arity}(f) \quad (2.12)$$

$$ge_{RM}(\xi, RM) = \texttt{\#'(lambda () } \xi\texttt{)} \quad (2.13)$$

Remarquons que la règle (2.7) est un cas particulier simplifié de la règle (2.8). Supposons en effet que le modèle exigé RM soit le modèle représenté par l'expression $p[t_1,\ldots,t_u \;/\; \phi_1,\ldots,\phi_v]$ et appliquons la règle (2.8) :

$$\begin{aligned} & cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(f.p[t_1,\ldots,t_u \;/\; \phi_1,\ldots,\phi_v](w_1,\ldots,w_n)) \\ = \; & \texttt{(f } cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(w_1) \ldots cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(w_n) \\ & \quad \texttt{(list } ge_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(\phi_1) \ldots ge_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(\phi_v)\texttt{))} \\ = \; & \texttt{(f } cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(w_1) \ldots cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(w_n) \\ & \quad \texttt{(list (nth 0 fm)} \ldots \texttt{(nth (1- } v\texttt{) fm)))} \\ = \; & \texttt{(f } cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(w_1) \ldots cp_{p[t_1,\ldots,t_u \,/\, \phi_1,\ldots,\phi_v]}(w_n) \texttt{ fm)} \end{aligned}$$

D'autre part, nous pourrions systématiquement, pour chaque fonction provenant d'une compilation, ajouter l'argument supplémentaire `fm`. Suivant cette convention, la valeur de `fm` serait toujours la liste vide dans le cas d'une fonction simulant un opérateur dont le modèle exigé ne comporte pas d'opérateurs formels. Dès lors, la règle (2.6) apparaît elle aussi comme un cas particulier de la règle générale (2.8), cas particulier dans lequel nous n'indiquons pas l'argument inutile.

D'après les mêmes considérations, on remarque, en ce qui concerne la fonction ge_{RM}, que les règles (2.10) et (2.11) sont des cas particuliers de la règle générale (2.12). La règle (2.13) rentre également dans ce cadre : elle ne fait que traduire que toute variable de sorte s présente dans une instanciation est assimilée à un opérateur de profil "$- \longrightarrow s$".

2.2.3 Simulation des opérateurs génériques par des fonctions du second ordre

Les sémantiques opérationnelles de FP2 et de Common Lisp étant respectivement déterminées par deux fonctions $eval_{FP2}$ (cf. §1.2.7) et $eval_{Lisp}$, il en découle notre critère de correction :

$$\boxed{(\forall t, \text{terme dont le modèle est "}\Diamond\text{"}), eval_{Lisp}(cp_{\Diamond}(t)) = eval_{Lisp}(cp_{\Diamond}(eval_{FP2}(t)))}$$

qui s'applique donc à tout terme générique de FP2, complètement instancié par rapport à "$\diamond$". Lorsque le terme t est composé uniquement de constructeurs[45], $eval_{FP2}(t) = t$, et le critère est toujours vérifié. Il acquiert tout son sens dans le cas contraire — présence d'opérateurs non constructeurs — où il signifie que la forme compilée du terme t et celle du résultat de l'évaluation symbolique de t — évaluation symbolique qui constitue la sémantique opérationnelle de FP2 — s'évaluent en la même expression Common Lisp. Examinons à présent un exemple.

Exemple 2.3 *Compilation de l'opérateur d'égalité des séquences génériques (voir sa spécification à l'exemple 0.88).*

```
(defun =S (x y fm)
  (if x
      (if y
          (if (funcall (first fm) (car x) (car y))
              (=S (cdr x) (cdr y) fm)))
      (endp y)))
```

Remarque Dans les tableaux précédents, nous avons désigné de façon générale les accès aux opérateurs formels par "`(nth ... fm)`", mais pour des raisons d'efficacité, le compilateur génère dans les cas usuels des formes telles que "`(first fm)`","`(second fm)`", ...

Comme nous l'avons signalé précédemment, nous ne traitons pas en détail la compilation du filtrage (cf. [164]) et invitons donc le lecteur à en vérifier "à la main" la correction sur cet exemple, compte tenu de la représentation des types *Bool* et *Seq*. Nous allons maintenant examiner un exemple d'exécution pour nous familiariser avec le traitement de la généricité. (D'autres exemples de codes générés sont donnés à l'annexe — cf. §B.3.)

Exemple 2.4 (Suite de l'exemple 2.3) *Le résultat de la compilation de l'expression :*

$$=_S.Equality[Seq[Seq[Nat]] \;/\; =_S.Equality[Seq[Nat] \;/\; =_S.Equality[Nat \;/\; =_N]]]([[[0]]], [[[0]]])$$

— cf. exemple 1.31 — est :

```
(=S (list (list (list 0)))
    (list (list (list 0)))
    (list #'(lambda (x1 x2)
              (=S x1 x2 (list #'(lambda (y1 y2)
                                  (=S y1 y2 (list #'=N))))))))
```

et s'évalue bien en la compilation du terme true, ainsi que la montre la trace suivante, à comparer avec l'évaluation qui figure à l'exemple 1.31.

(Nous ne reproduisons ci-après que la trace des fonctions Common Lisp qui implantent les opérateurs "$=_S$" et "$=_N$", c'est-à-dire que nous donnons, pour chacun de leurs appels, leurs arguments et la valeur retournée.)

[45] En supposant, pour les constructeurs partiels, que les arguments sont licites — cf. chapitre 4.

```
===> (=S (list (list (list 0)))
         (list (list (list 0)))
         (list #'(lambda (x1 x2)
                     (=S x1 x2 (list #'(lambda (y1 y2)
                                           (=S y1 y2 (list #'=N))))))))

     ===> (=S (list (list 0))
              (list (list 0))
              (list #'(lambda (y1 y2)
                          (=S y1 y2 (list #'=N)))))

          ===> (=S (list 0) (list 0) (list #'=N))

               ===> (=N 0 0)

               <=== T

               ===> (=S nil nil (list #'=N))

               <=== T

          <=== T

          ===> (=S nil nil (list #'(lambda (y1 y2)
                                       (=S y1 y2 (list #'=N)))))

          <=== T

     <=== T

     ===> (=S nil nil (list #'(lambda (x1 x2)
                                  (=S x1 x2 (list #'(lambda (y1 y2)
                                                        (=S y1 y2 (list #'=N))))))))

     <=== T

<=== T
```

Abordons le problème plus formellement, après la donnée de cet exemple. Nous préoccupant plus particulièrement de la compilation des opérateurs génériques dont la propriété exigée comporte des opérateurs dans sa signature, nous allons admettre la correction de la compilation du filtrage (cf. [164]), et la correction de la compilation des opérateurs dont la propriété exigée est "◇" ou n'admet pas d'opérateurs dans sa signature.

Théorème 2.5 *La compilation des opérateurs génériques de FP2 qui admettent des opérateurs formels est correcte.*

Preuve L'ordre d'apparition des opérateurs formels dans le modèle exigé étant fixé, chaque opérateur formel est dès lors pleinement représenté par son rang dans la forme compilée du modèle que contient l'argument `fm`. C'est ce que traduisent les règles (2.6) et (2.9).

Passons à présent aux termes complètement instanciés et examinons une évaluation quelconque d'une fonction Common Lisp qui provient de la compilation d'un opérateur générique. Dès lors que le terme est complètement instancié, le modèle contient les opérateurs effectifs, qualifiés par leur instanciation. Considérons donc un opérateur formel, noté ϕ, occupant le rang i parmi les opérateurs du modèle exigé, et instancié. Remarquons tout d'abord que si cet opérateur est instancié par une variable, la règle (2.13) assure la correction au niveau opérationnel. Étudions maintenant le cas d'une instanciation par un opérateur générique f lui-même qualifié par l'expression de modèle M_f. Le terme à évaluer étant par hypothèse complètement instancié par rapport à "$\diamond$", si l'opérateur formel ϕ est lié à un autre opérateur formel ϕ', nécessairement ϕ' est lui-même complètement instancié par rapport à "$\diamond$". C'est pourquoi on peut supposer sans perte de généralité que ϕ est directement lié à $f.M_f$. Soit :

$$\phi(w_1, \ldots, w_n)$$

le terme à évaluer, où n est l'arité de ϕ. En accord avec la stratégie d'évaluation par valeur de FP2 (cf. §1.2.7), nous convenons sans perte de généralité que les arguments $w_1, \ldots, w_n$ sont en forme normale. Ce que nous allons prouver, c'est que l'évaluation de la forme compilée du terme $\phi(w_1, \ldots, w_n)$ se ramène bel et bien à l'évaluation du terme :

$$f.M_f(w_1, \ldots, w_n)$$

Au §1.2.7, nous avions traduit cette propriété par la règle (1.9). Nous allons raisonner par récurrence sur la profondeur du modèle M_f. Posons, pour une expression de modèle M quelconque :

$$\begin{array}{rcll} depth(M) & = & 0 & \neg operators\text{-}in(M) \\ depth(p[t_1, \ldots, t_u \ / \ \tilde{g}_1, \ldots, \tilde{g}_v]) & = & 1 + \max\limits_{1 \leqslant j \leqslant v} op\text{-}depth(\tilde{g}_j) & \end{array}$$

où chaque $\tilde{g}_j$ $(1 \leqslant j \leqslant v)$ représente soit un opérateur formel, soit un opérateur générique qualifié par une expression de modèle. La fonction *op-depth* est définie comme suit :

$$\begin{array}{rcll} op\text{-}depth(g.M_g) & = & depth(M_g) & \\ op\text{-}depth(\phi) & = & op\text{-}depth(\tilde{g}) & ([\phi \mapsto \tilde{g}]) \end{array}$$

g étant un opérateur générique et M_g une expression de modèle qui le qualifie. La dernière égalité traduit le même raisonnement que précédemment : les termes à évaluer étant par hypothèse complètement instanciés par rapport à "$\diamond$", tout opérateur formel ϕ est lié soit à un opérateur générique qualifié par une expression représentant un modèle complètement instancié, soit à un opérateur formel lui-même complètement instancié.

- Cas $depth(M_f) = 0$

  D'après la règle (2.6), nous avons :

  $$cp_{RM}(\phi(w_1, \ldots, w_n)) = \texttt{(funcall (nth } position_{RM}(\phi) \texttt{ fm) } cp_{RM}(w_1) \ \ldots \ cp_{RM}(w_n)\texttt{)}$$

  RM étant le modèle exigé du module où est défini l'opérateur dont nous étudions une évaluation. Si cette évaluation est effectuée avec une expression de modèle dont l'opérateur générique de rang i — avec $i = position_{RM}(\phi)$ — est $f.M_f$, alors d'après la règle (2.10) :

  $$ge_{RM}(f.M_f) = \texttt{\#'f}$$

Par conséquent, l'expression :

$$\texttt{(funcall (nth } i \texttt{ fm) w}_1 \texttt{ ... w}_n\texttt{)} \quad (2.14)$$

s'évalue en :

$$\texttt{(funcall \#'f w}_1 \texttt{ ... w}_n\texttt{)}$$

c'est-à-dire :

$$\texttt{(f w}_1 \texttt{ ... w}_n\texttt{)}$$

ce qui correspond bien à l'appel de l'opérateur `f`. Rappelons que nous admettons la correction de la compilation des opérateurs dont la propriété exigée est "◇" ou n'admet pas d'opérateurs dans sa signature. Il s'ensuit que la compilation des termes génériques et des expressions de modèles est correcte dans ce cas. □

- Passage au successeur

Posons maintenant :

$$M_f = p[t_1, \ldots, t_u \;/\; \tilde{f}_1, \ldots, \tilde{f}_v]$$

tel que $depth(M_f) = q$, avec $q > 0$, et supposons à présent que l'évaluation de tout opérateur formel instancié par un opérateur générique g qualifié par une expression de modèle M_g avec $depth(M_g) < q$ est correcte. D'après la règle (2.12), l'opérateur générique de rang i dans l'expression de modèle est compilé comme suit :

$$ge_{RM}(f.p[t_1, \ldots, t_u \;/\; \tilde{f}_1, \ldots, \tilde{f}_v]) =$$
$$\texttt{\#'(lambda (x}_1 \texttt{ ... x}_n\texttt{) (f x}_1 \texttt{ ... x}_n \texttt{ (list } ge_{RM}(\tilde{f}_1) \texttt{ ... } ge_{RM}(\tilde{f}_v)\texttt{)))}$$

L'expression (2.14) se développe cette fois en :

$$\texttt{(funcall \#'(lambda (x}_1 \texttt{ ... x}_n\texttt{) (f x}_1 \texttt{ ... x}_n \texttt{ (list } ge_{RM}(\tilde{f}_1) \texttt{ ... } ge_{RM}(\tilde{f}_v)\texttt{)))}$$
$$\texttt{w}_1 \texttt{ ... w}_n\texttt{)}$$

et s'évalue en :

$$\texttt{(f w}_1 \texttt{ ... w}_n \texttt{ (list } ge_{RM}(\tilde{f}_1) \texttt{ ... } ge_{RM}(\tilde{f}_v)\texttt{))}$$

où l'expression "`(list` $ge_{RM}(\tilde{f}_1)$ `...` $ge_{RM}(\tilde{f}_v)$`)`" est donc la valeur à lier à l'argument `fm` lors de l'appel de la fonction Common Lisp `f`. Chaque $\tilde{f}_j$ $(1 \leqslant j \leqslant v)$ est :

⋆ soit un opérateur générique instancié ;

⋆ soit un opérateur formel présent dans RM et représenté par son rang : d'après l'hypothèse, cet opérateur formel est instancié par un opérateur générique complètement instancié.

Dans les deux cas, l'opérateur générique est par hypothèse qualifié par une expression de modèle dont la profondeur est strictement inférieure à q. D'après l'hypothèse de récurrence, tout appel d'un opérateur formel instancié par un opérateur générique qualifié par une expression de modèle dont la profondeur est strictement inférieure à q est correct. On en déduit que l'évaluation de l'instanciation de l'opérateur formel ϕ par $f.M_f$ est correcte. □

Cette preuve achève l'étude de la compilation en Common Lisp des opérateurs et des termes génériques de FP2.

Chapitre 3

Raccourcis de notation

Jusqu'à présent, nous avons considéré les bases de la partie fonctionnelle en utilisant des formulations univoques, mais parfois (souvent !) lourdes. Nous allons aborder maintenant les diverses facilités offertes aux utilisateurs, montrer comment elles se ramènent aux notations complètes. Ces facilités ont été introduites dans LPG, de façon relativement informelle [17, 19]. Nous présentons d'abord ces raccourcis de manière intuitive, sur des exemples, et nous donnons ensuite une formulation rigoureuse des outils et des règles de l'analyse sémantique.

3.1 Approche intuitive

3.1.1 Instanciation implicite

Reprenons l'exemple de l'opérateur d'égalité sur les séquences génériques. Nous considérons l'expression $=_S([0],[0])$ et supposons connue la déclaration de modèle :

model *EQNAT* **is** $Equality[Nat\ /\ =_N]$
endmodel

"$=_S$" a pour profil $Seq[t] \times Seq[t] \rightarrow Bool$, le modèle exigé étant $Equality[t\ /\ eq]$. L'analyse des arguments de "$=_S$" fait intervenir la substitution $[t \mapsto Nat]$. Si *EQNAT* est le *seul* modèle déclaré de l'égalité sur le type *Nat*, alors par unification[46] de $Equality[Nat\ /\ =_N]$ et de $Equality[t\ /\ eq]$ avec $[t \mapsto Nat]$, on en déduit $[eq \mapsto =_N]$. Nous venons d'instancier *implicitement* l'opérateur "$=_S$".

De même, considérons l'expression $=_S([[0]],[[0]])$ et les déclarations des modèles *EQNAT* et *EQSEQ* :

model *EQSEQ* **req** $Equality[t_0\ /\ eq_0]$ **is** $Equality[Seq[t_0]\ /\ =_S]$
endmodel

Notons à nouveau $Equality[t\ /\ eq]$ le modèle exigé. Cette fois, l'analyse des arguments de "$=_S$" fait intervenir la substitution $[t \mapsto Seq[Nat]]$. Si *EQSEQ* est le seul modèle déclaré de l'égalité sur les séquences génériques, alors par unification de $Equality[Seq[t_0]\ /\ =_S]$ et de $Equality[t\ /\ eq]$ avec $[t \mapsto Seq[Nat]]$, nous obtenons $[eq \mapsto =_S]$ et $[t_0 \mapsto Nat]$. Il nous faut maintenant chercher un modèle

[46] Au §3.3.2, nous définirons plus précisément l'unification entre expressions représentant des modèles. Pour l'instant, comprenons qu'il s'agit d'une résolution d'équation en vue de substituer les paramètres formels par des paramètres effectifs.

de $Equality[t_0\ /\ eq_0]$ avec $[t_0 \mapsto Nat]$. Comme précédemment, un tel modèle est $EQNAT$ et nous obtenons $[eq_0 \mapsto =_N]$. En reportant toutes ces substitutions, l'instance cherchée est donc :

$$=_S.Equality[Seq[Nat]\ /\ =_S.Equality[Nat\ /\ =_N]]$$

Nous avons donc vu que la seule donnée des sortes effectives peut suffire dans les cas où l'on peut déduire les opérateurs des déclarations de modèles. On étend sans difficulté cette possibilité aux expressions de sortes, de modèles, ainsi qu'aux présentations de processus. Ainsi, si — par exemple — *NR-Seq* désigne les séquences génériques sans répétition d'éléments — ce type exigeant la propriété *Equality* —, *NR-Seq*[*Nat*] représentera sans ambiguïté *NR-Seq.EQNAT*.

Bien sûr, l'instanciation implicite n'est possible que si l'on peut inférer un modèle et un seul. Considérons la propriété d'ordre total et le type *O-Tree* des arbres binaires ordonnés (cf. annexe, exemple B.16). Si *INCNAT* (ordre croissant, cf. §1.2.4) est le seul modèle de cette propriété déclaré sur le type *Nat*, alors *O-Tree*[*Nat*] désignera la sorte *O-Tree.INCNAT*. Si sont définis à la fois *INCNAT* et *DECNAT*, alors *O-Tree*[*Nat*] est une expression de sorte ambiguë.

Remarque Étant donné que pour chaque type, il n'existe qu'un seul modèle de *Ftype*, l'instanciation implicite de *Ftype* est toujours possible. On justifie ainsi la notation provisoire — *Seq*[*Nat*], *Seq*[*Seq*[*Nat*]], ... — adoptée dans les chapitres précédents.

Note En FP2, l'utilisateur est complètement déchargé de l'indication des modèles de *Ftype*. À la lecture de la spécification d'un module de type T exigeant le modèle M, tout se passe comme si FP2 "ajoutait" la définition suivante :

model req M **is** $Ftype[T.M\ /\ ]$
endmodel

Ceci confère à FP2 la légèreté des langages fondés sur le polymorphisme, c'est-à-dire des langages dont les fonctions admettent des profils comprenant des types variables. (Si l'utilisateur spécifie lui-même un modèle de *Ftype*, FP2 "sait" que cela n'introduit pas d'ambiguïté.)

De même pour une propriété comportant dans sa signature des sortes :

prop $P[t_1, \ldots, t_n\ /\ \ldots]$
⋮
endprop

Si l'utilisateur ne les spécifie pas, FP2 "ajoute" les spécifications d'inclusion suivantes :

$$P[t_1, \ldots, t_n\ /\ \ldots]\ \textbf{includes}\ Ftype[t_i\ /\ ], (1 \leqslant i \leqslant n)$$

3.1.2 Instanciation explicite simplifiée

Elle consiste, au lieu d'indiquer le modèle dans l'appel de l'opérateur générique, à n'indiquer que les opérateurs effectifs. Par exemple : $=_S[=_N]([0],[0])$. Étant donné le modèle exigé $Equality[t\ /\ eq]$, la substitution $[t \mapsto Nat]$ se déduit de l'analyse des arguments et la substitution $[eq \mapsto =_N]$ de l'instanciation explicite. Cette facilité d'instanciation est très utilisée lorsqu'on simule à l'aide d'opérateurs génériques les opérateurs "du deuxième ordre" inspirés d'APL ou de FP [4] (cf. l'opérateur *alpha* — exemple 3.2).

Il résulte de tout ceci une grande liberté dans les choix possibles des notations. Ainsi l'expression :

$$=_S.EQSEQ.EQSEQ.EQNAT([[[0]]],[[[0]]])$$

qui, comme nous l'avons vu, se développe par composition de morphismes, en :

$$=_S.Equality[Seq[Seq[Nat]] \;/\; =_S.Equality[Seq[Nat] \;/\; =_S.Equality[Nat \;/\; =_N]]]([[[0]]],[[[0]]]) \quad (3.1)$$

peut encore s'écrire des façons suivantes :

$$=_S([[[0]]],[[[0]]])$$

$$=_S.EQSEQ.EQSEQ\overbrace{[Nat]}^{(1)}([[[0]]],[[[0]]])$$

$$=_S.EQSEQ\overbrace{[Seq[Nat]]}^{(1)}([[[0]]],[[[0]]])$$

$$=_S[=_S[=_S[=_N]]]([[[0]]],[[[0]]])$$

$$=_S.Equality[Seq[Seq[Nat]] \;/\; =_S\overbrace{[=_S\underbrace{\quad}_{(3)}]}^{(2)}]([[[0]]],[[[0]]])$$

$$=_S.Equality[Seq[Seq[Nat]] \;/\; =_S\underbrace{\quad}_{(3)}]([[[0]]],[[[0]]])$$

$$=_S\overbrace{[=_S\underbrace{\quad}_{(3)}]}^{(2)}([[[0]]],[[[0]]])$$

(1) : instanciation implicite de *EQSEQ*.
(2) : instanciation explicite de "$=_S$".
(3) : instanciation implicite de "$=_S$".

Il doit cependant apparaître clairement que la multiplicité des notations pour désigner une même instanciation doit être comprise comme une liberté fournie par le langage : *toute* sorte, ainsi que *tout* modèle ou *toute* présentation de processus, quelque soit son occurrence dans une expression, peut être instanciée complètement, ou implicitement à condition que l'on puisse déduire l'instanciation complète. De même, *tout* opérateur peut être instancié complètement, implicitement, ou explicitement. Dans le cas précédent, l'analyse de toutes les expressions données les ramène à la forme (3.1).

3.1.3 Analyse de la sorte d'une expression — Ambiguïtés

Des diverses formes de polymorphisme présentes en FP2, il résulte que la sorte d'une expression peut parfois être équivoque. Commençons par examiner un cas de polymorphisme *ad hoc*.

Exemple 3.1 *Définissons les types French-Colours et Italian-Colours comme suit :*

type *French-Colours*
  cons *blue, white, red* : –
endtype

type *Italian-Colours*
  cons *green, white, red* : –
endtype

L'expression *white* est ambiguë : si dans la boucle d'évaluation de FP2, on désire spécifier un tel terme, on doit préciser la sorte : "*white* : *French-Colours*" ou "*white* : *Italian-Colours*".

Une analyse sémantique stricte qui traite toute sous-expression indépendamment du contexte nécessite la mention de la sorte à chaque utilisation d'un tel terme. Nous pouvons remarquer cependant, que si l'on définit un opérateur f de domaine *French-Colours*, et qu'il n'existe aucun opérateur f de domaine *Italian-Colours*, alors l'expression $f(white)$ ne peut *que* désigner :

$$f(white : French\text{-}Colours)$$

Examinons un deuxième cas d'ambiguïté dû au polymorphisme paramétrique. Une expression telle que *nil* est ambiguë parce que *nil* peut désigner aussi bien une séquence vide de booléens qu'une séquence vide d'entiers ou encore une séquence vide d'éléments de sorte t où t est une sorte formelle de l'environnement générique. Comme précédemment, si l'on désire évaluer une séquence vide, on doit spécifier son type : “*nil* : *Seq*[*Nat*]” ou “*nil* : *Seq*[*Bool*]”. L'ambiguïté provient cette fois de ce qu'il existe un paramètre générique dans le codomaine qui est absent dans le domaine. Par contre, si nous écrivons “0 <+ *nil*”, et que “<+” désigne de façon univoque l'ajout en tête de séquence, cette expression ne peut avoir de sens *que* si *nil* désigne “*nil* : *Seq*[*Nat*]”. C'est-à-dire que nous avons *inféré* la sorte de *nil* dans l'expression 0 <+ *nil*.

Remarque La notation “$x : t$” signifie que t est la *sorte* de l'expression x. Dans “$f.M(x_1, \ldots, x_n)$”, M désigne le *modèle* de f. À titre d'exemple, “*nil* : *Seq*[*Nat*]”, “*nil.Ftype*[*Nat* /]”, “*nil.Ftype*[*Nat* /] : *Seq*[*Nat*]” sont trois expressions équivalentes.

Les deux cas d'ambiguïtés se résolvent différemment : dans le premier cas, à chaque exemplaire différent d'un symbole surchargé, correspond une fonction Lisp différente. Néanmoins, cela montre que, dans un cas comme dans l'autre, l'analyse du contexte est nécessaire pour lever de telles ambiguïtés.

Ces deux cas d'ambiguïté peuvent se rencontrer, comme va le montrer l'exemple suivant :

Exemple 3.2 *Définissons un opérateur alpha inspiré de l'opérateur de même nom, présent en FP.*

```
prop Formal-Op[t1, t2 / f]
   opns f : t1 → t2
endprop

enr req Formal-Op[t1, t2 / f]
   opns alpha : Seq[t1] → Seq[t2]
   vars x     : t1
        s     : Seq[t1]
   rules
      <>      alpha(nil)   ==>  nil
      <>  alpha(x <+ s)    ==>  f(x) <+ alpha(s)
endenr
```

Là aussi, nous nous trouvons dans un cas où un paramètre générique est présent dans le codomaine et absent dans le domaine. Mais la mention de l'opérateur effectif peut faire déduire ce paramètre :

$$alpha[succ]([0])$$

Si le seul profil de *succ* est $Nat \rightarrow Nat$, alors l'instanciation est :

$$Formal\text{-}Op[Nat, Nat \ / \ succ]$$

Par contre, soit un opérateur “♯” surchargé comme suit :

$$\sharp \ : \ Nat \rightarrow Nat$$
$$\sharp \ : \ Nat \rightarrow Bool \qquad (3.2)$$

alors l'expression *alpha*[♯]([0]) est ambiguë. Par contre, l'expression “*not alpha*[♯]([0])” ne l'est plus et “♯” y désigne l'exemplaire (3.2).

3.1.4 Conclusion de l'approche intuitive

Si nous désirons offrir à l'utilisateur d'une part la possibilité de ne préciser la sorte d'une expression que lorsque c'est vraiment indispensable, d'autre part la facilité de donner l'instanciation d'un opérateur sous la forme la plus légère possible, nous sommes amené à différencier une expression considérée globalement, que nous appellerons **expression complète**, de ses sous-expressions propres. La règle adoptée est que l'analyse d'une expression complète doit être univoque. Certains sous-termes peuvent admettre plusieurs analyses sémantiques, mais par recoupements successifs (que l'on peut comprendre comme des résolutions d'équations), un seul "terme sémantique", sans sous-terme ambigu, doit subsister à la fin de l'analyse. Lorsque nous sommes dans la boucle d'évaluation, cela revient à un mécanisme d'inférence d'expressions de sortes et d'expressions de modèles. Lorsque nous nous trouvons à l'intérieur d'une déclaration de module, on utilisera l'information contenue dans les déclarations des profils des opérateurs et des sortes des variables.

Informellement, la règle d'analyse est simple à formuler. On peut cependant remarquer la multiplicité des sources d'information pour déduire les sortes et les modèles. C'est la façon dont nous allons utiliser toutes ces sources que nous allons spécifier formellement, après avoir fixé les conditions de surcharge.

3.2 Conditions d'utilisation de la surcharge

Il est certain que la surcharge complique l'analyse des termes, même si elle n'est que l'extension aux langages informatiques d'un procédé très naturel en Mathématiques qui est de désigner par le même symbole des opérations similaires. Une solution radicale serait de ne pas offrir la possibilité de surcharge (elle est absente de ML [89], KRC [173], MIRANDA [174]). Une autre solution, plus douce, serait de ne la permettre que lorsque le domaine suffit à lever l'ambiguïté (c'est la convention de μFP2 [161] et de LUSTRE [38]). Malheureusement, l'étude des cas d'ambiguïtés dus aux paramètres génériques montre qu'il existe des cas, même très simples, où la connaissance du domaine ne suffit pas, par exemple le terme *nil*. Nous considérerons donc que les ambiguïtés dues à la surcharge pourront être levées aussi bien par le domaine que par le codomaine.

La seule restriction est qu'il ne doit pas exister deux profils unifiables (en remplaçant les paramètres formels des profils par des variables que nous qualifierons de *variables de travail*) parmi les divers exemplaires d'un symbole surchargé. Ainsi :

$$\begin{array}{lrcll} f : & Seq[t] & \rightarrow & Seq[t] & (Ftype[t\ /\ ]) \\ f : & Seq[Nat] & \rightarrow & Seq[Bool] & \end{array}$$

constitue une surcharge licite du symbole f, mais non :

$$\begin{array}{lrcll} f : & Seq[t] & \rightarrow & Seq[t] & (Ftype[t\ /\ ]) \\ f : & Seq[Nat] & \rightarrow & Seq[Nat] & \end{array}$$

Note On pourrait en fait retenir la plus petite substitution (au sens "borne inférieure" d'un treillis) et considérer que les cas où les signatures sont unifiables sans être identiques (à un renommage près des paramètres génériques) correspondent à des *redéfinitions* d'opérateurs dans des cas particuliers. C'est l'approche de LPG 1.8. Ainsi, soient les déclarations suivantes :

$$f : \quad Seq[Nat] \rightarrow Seq[Nat] \tag{3.3}$$

$$f : \quad Seq[t] \rightarrow Seq[t] \quad (Ftype[t\ /\ ]) \tag{3.4}$$

$$f : \quad Seq[Seq[t]] \rightarrow Seq[Seq[t]] \quad (Ftype[t\ /\ ]) \tag{3.5}$$

$$f([0]) \quad \text{fait référence à l'exemplaire} \quad (3.3)$$
$$f([true]) \quad \ldots \quad (3.4)$$
$$f([[true]]) \quad \ldots \quad (3.5)$$

Nous avons préféré que les signatures soient disjointes.

La philosophie sous-jacente est que nous ne désirons pas *cacher* d'autres opérateurs que ceux que l'utilisateur déclare explicitement privés. Dans l'exemple précédent, l'opérateur f spécifié en (3.3) cache celui qui est déclaré en (3.4) dans le cas de l'application à une séquence d'entiers naturels, et ce dernier devient dès lors *inaccessible* pour une telle application.

3.3 Définition du formalisme

Dans le chapitre 1, nous avons utilisé les expressions de modèles afin de noter les morphismes de présentations sous une forme plus légère, en utilisant l'ordre lexical. Les buts esquissés au début de ce chapitre ont fait ressortir la nécessité d'un calcul sur les expressions de modèles, de sortes, et d'opérateurs, et c'est ce que nous allons définir à présent.

3.3.1 Syntaxe

Lors de l'approche intuitive, nous avons ressenti le besoin de considérer les expressions représentant des modèles comme des termes, de résoudre des équations entre expressions de modèles. Nous définissons donc le **langage des expressions de modèles** T_m, le **langage des expressions de sortes génériques** T_t, et le **langage des expressions d'opérateurs génériques** T_ω. Ceci va nous permettre d'exprimer formellement les opérations nécessaires aux calculs d'instanciations et à l'analyse sémantique.

Soient S un ensemble de sortes et V_t un ensemble de "sortes variables", Ω un ensemble d'opérateurs et V_ω un ensemble d'"opérateurs variables", les éléments de Ω et de V_ω étant indicés par des éléments de $T_t^* \times T_t$. Soit également Π un ensemble de symboles de propriétés, ensemble qui contient le symbole "$\diamond$". Une première approche des ensembles de mots T_m, T_t, et T_ω est fournie par les inclusions suivantes :

$$\begin{aligned} T_m &\subseteq \Pi.(T_t^*.T_\omega^*) \\ T_t &\subseteq S.T_m \cup V_t \\ T_\omega &\subseteq \Omega.T_m \cup V_\omega \end{aligned}$$

Par convention, nous noterons un élément de $\Pi.(T_t^*.T_\omega^*)$ par :

$$p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n]$$

Il nous reste alors à caractériser les expressions *bien formées* à l'intérieur des ensembles $\Pi.(T_t^*.T_\omega^*)$, $S.T_m \cup V_t$, et $\Omega.T_m \cup V_\omega$. Dans le cas de l'algèbre des termes (cf. définition 0.6), un terme tel que $\omega(t_1, \ldots, t_n)$ est bien formé si l'arité de l'opérateur ω et les sortes des termes t_i comparés aux sortes du domaine de ω sont respectés. Dans le langage des expressions de modèles, nous devons certes refuser des expressions telles que : $Ftype[Nat, Bool\ /\ ]$ — car la signature ordonnée de $Ftype$ ne comporte qu'une seule sorte —, mais aussi $Equality[Nat\ /\ +]$, car "+" est de profil $(Nat \times Nat) \rightarrow Nat$ et non $(Nat \times Nat) \rightarrow Bool$. Pour définir les expressions de modèles bien formées, nous allons les dériver à partir d'expressions primitives, obtenues par des morphismes de signatures.

Définition 3.3 *Soient $p \in \Pi$, un symbole de propriété, et $\Sigma_p = (S_p, \Omega_p)$ la signature* ordonnée *de p. On appelle* **expression primitive** *de la propriété p toute image de Σ_p dans (V_t, V_ω) par un morphisme de signatures* injectif.

Les expressions primitives d'une propriété p étant identiques à un renommage bijectif des variables près, nous noterons simplement $prim(p)$ l'une quelconque d'entre elles.

Étant donné une telle expression μ, nous désignerons par $\pi(\mu)$ le symbole de propriété en tête :

$$\pi(p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n]) \stackrel{\text{déf}}{=} p$$

"$\diamond[\ /\ ]$", "$s.\diamond$", "$f.\diamond$" seront notés plus simplement par "$\diamond$", "s" et "f".

Nous allons maintenant définir les substitutions sur les expressions de modèles, et nous les utiliserons pour dériver toutes les expressions bien formées à partir d'une expression primitive.

De même que pour les termes d'une algèbre libre, on définit une substitution par la donnée des images des variables pour lesquelles elle n'équivaut pas à l'identité, de même, sur les expressions de modèles, on définira une substitution σ par la donnée des images non identiques des sortes variables et des opérateurs variables, c'est-à-dire par un couple $(\sigma_{V_t}, \sigma_{V_\omega})$, où $\sigma_{V_t} : V_t \rightarrow T_t$ et $\sigma_{V_\omega} : V_\omega \rightarrow T_\omega$. σ est alors complètement définie au moyen des règles suivantes :

$$\sigma(p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n]) = p[\sigma(t_1), \ldots, \sigma(t_m) \ / \ \sigma(f_1), \ldots, \sigma(f_n)]$$

$$\begin{aligned} \sigma(x_t) &= \sigma_{V_t}(x_t) \qquad (x_t \in V_t) \\ \sigma(s.\mu) &= s.\sigma(\mu) \end{aligned}$$

$$\begin{aligned} \sigma(x_\omega : t_1 \times \cdots \times t_m \rightarrow t) &= \sigma_{V_\omega}(x_\omega) : \sigma(t_1) \times \cdots \times \sigma(t_m) \rightarrow \sigma(t) \quad (x_\omega \in V_\omega) \\ \sigma((\phi : t_1 \times \cdots \times t_n \rightarrow t).\mu) &= (\phi : \sigma(t_1) \times \cdots \times \sigma(t_n) \rightarrow \sigma(t)).\sigma(\mu) \end{aligned}$$

Cette notion de substitution s'étend aux termes donnés avec leur sorte. Si "$u : s$" désigne le terme u de sorte s, alors :

$$\sigma(u : s) = u : \sigma(s)$$

De même en ce qui concerne un terme formé d'un opérateur ω donné avec le modèle qui le qualifie :

$$\begin{aligned} \sigma(\omega.M) &= \omega.\sigma(M) \qquad (\text{arity}(\omega) = 0) \\ \sigma(\omega.M(u_1, \ldots, u_n)) &= \omega.\sigma(M)(u_1, \ldots, u_n) \qquad (\text{arity}(\omega) > 0) \end{aligned}$$

Nous pouvons à présent caractériser les mots qui sont des éléments de T_m :

Soit $\mu \in \Pi.(T_t^* \times T_\omega^*)$, $\mu = p[t_1, \ldots, t_m \ / \ f_1, \ldots, f_n]$.

$$\boxed{\mu \in T_m \stackrel{\text{déf}}{\Longleftrightarrow} (\exists \sigma), \mu = \sigma(prim(p))}$$

D'une façon plus concise : $\mu \in T_m \Longleftrightarrow (\exists \sigma), \mu = \sigma(prim(\pi(\mu)))$.

Remarque Nous ne nous préoccupons pas de la validité des équations d'une propriété, et travaillons uniquement au niveau des morphismes de signatures. Ce point sera commenté au §3.3.4.

Nous caractérisons maintenant les mots de T_m et de T_ω en traduisant que le symbole de propriété du modèle doit coïncider avec le symbole représentant la propriété exigée par le type ou l'opérateur.

$$\boxed{\tau \in T_t \Longleftrightarrow (\tau \in V_t) \vee (\tau = s.\mu, s \in S, \mu \in T_m, \text{prop-exigée}(s) = \pi(\mu))}$$

$$\boxed{f \in T_\omega \Longleftrightarrow (f \in V_\omega) \vee (f = \phi.\mu, \phi \in \Omega, \mu \in T_m, \text{prop-exigée}(s) = \pi(\mu))}$$

Exemple 3.4 *Soient $t_1 \in V_t$ et $f_1, f_2 \in V_\omega$, les profils étant :*

$$f_1, f_2 : t_1 \times t_1 \to Bool$$

Les deux expressions de modèles suivantes sont des expressions primitives :

$$\begin{array}{c} Equality[t_1 \;/\; f_1 \quad] \\ Total\text{-}Order[t_1 \;/\; f_1, f_2] \end{array}$$

Les expressions suivantes sont des expressions de modèles qui appartiennent à T_m :

$$\begin{array}{c} Equality[Nat \;\;/ =_N \quad] \\ Total\text{-}Order[Nat \;\;/ f_1, =_N] \\ Total\text{-}Order[Bool \;/ \Leftrightarrow, f_2 \;] \end{array}$$

On a pu remarquer que nous n'avons pas inclus dans notre syntaxe les expressions de modèles génériques vues au §1.2.5. Nous supposons qu'elles sont développées suivant le procédé que nous avons décrit à ce moment.

3.3.2 Opérations

Nous souhaitons pouvoir effectuer des résolutions d'équations entre éléments de T_m, entre éléments de T_t, et entre éléments de T_ω. Le premier pas vers cet objectif est d'étendre le concept d'unification aux expressions de modèles.

Définition 3.5 *Deux expressions de modèles μ_1 et μ_2 sont unifiables si et seulement s'il existe une substitution σ telle que $\sigma(\mu_1) \equiv \sigma(\mu_2)$.*

Comme dans la définition 0.14, "$\equiv$" désigne une égalité syntaxique : ici, c'est une égalité syntaxique entre expressions de modèles.

En disposant différemment les divers sous-termes et en ajoutant des opérateurs auxiliaires, il est possible de décrire les expressions de modèles, de sortes et d'opérateurs sous une présentation analogue aux termes d'une algèbre des termes, c'est-à-dire soit une variable, soit un symbole qui n'est pas une variable et qui est suivi suivi d'une liste éventuellement vide d'expressions. Il s'ensuit que l'unification entre expressions de modèles est un problème exactement équivalent à l'unification entre termes d'une algèbre libre. On peut donc dégager la notion de plus petit unificateur, utiliser la même notation :

$$\mu_1 \overset{\sigma}{\approx} \mu_2$$

Nous définissons également l'unification entre expressions de sortes et entre expressions d'opérateurs, ainsi que le filtrage avec des notations tout à fait analogues. De même, nous qualifierons d'expressions **fermées** les expressions de modèles, de sortes, et d'opérateurs ne comportant pas de variables de travail appartenant à V_t ou V_ω. Une substitution qui conduit à une expression de modèle fermée sera également appelée substitution **fermée**.

3.3.3 Sémantique

Nous conviendrons qu'une expression μ de T_m désigne des expressions de modèles qui sont des instanciations *possibles* de la propriété exigée. Une telle instanciation est obtenue en appliquant à μ une substitution fermée quelconque. Dans le cas où μ est une expression primitive, μ représente sémantiquement *toutes* les instanciations possibles de la propriété exigée. Si μ est une expression fermée, μ représente *une* instanciation de la propriété exigée.

Soit une substitution $\sigma : T_m \to T_m$, telle que $\sigma(\mu_1) = \mu_2$, avec $\mu_1, \mu_2 \in T_m$. Il résulte de la sémantique précédente pour les expressions de T_m qu'elle représente une rétraction[47] des expressions de modèles que représente μ_1 vers les expressions de modèles que représente μ_2.

3.3.4 Implantation

Nous n'allons pas décrire en détail l'implantation des structures et des outils que nous venons d'introduire, pas plus que nous ne décrirons l'implantation des règles d'analyse données au §3.5, mais nous signalons toutefois qu'elle est plus performante que la stricte transposition des règles données.

Ce qu'il est important de remarquer, c'est qu'à beaucoup de notions précédentes correspondent en réalité des concepts d'implantation assez simples. Ainsi, une expression primitive d'une propriété s'obtient par *copie* du graphe Common Lisp qui représente la signature ordonnée de cette propriété, en remplaçant les sortes et les opérateurs de la signature par des variables de travail qui devront être substituées avant la fin du calcul. De même, nous avons déjà mentionné (cf. §2.1.1) qu'à chaque exemplaire d'un opérateur surchargé était attribué un nom interne unique, accès à une fonction Common Lisp. Nous verrons au §3.5.1 comment sont levés les ambiguïtés dans les termes des équations et dans les termes à évaluer. En ce qui concerne les expressions de T_m, la mention du profil rend l'opérateur univoque lorsqu'il apparaît dans une expression comme :

$$Equality[Nat \ / \ = : Nat \times Nat \to Bool]$$

que nous avons pris l'habitude de noter plus légèrement par :

$$Equality[Nat \ / \ =_N]$$

Au §1.2.6, nous avons indiqué que pour chaque instanciation qu'il écrit, c'était à l'utilisateur d'assumer que les équations d'une propriété sont satisfaites ou de mener leur démonstration à l'aide d'un outil de preuve. Compte tenu de cette convention, il est tout à fait loisible de considérer uniquement des opérations sur les signatures au niveau du calcul d'instanciations, uniquement des morphismes de signatures au niveau sémantique.

3.4 Le problème posé

On peut considérer que les sortes et les modèles fournis par l'utilisateur le sont d'une manière *incomplète*. En ce qui concerne les expressions de modèles, il peut fournir uniquement les sortes (par exemple, *O-Tree*[*Nat*]), ou celles-ci peuvent être déduites de l'analyse du domaine et du codomaine de l'opérateur. Ce que nous allons spécifier, c'est comment trouver le couple d'instanciation d'un opérateur ou d'un type, à l'aide d'un modèle incomplet. Les arguments non spécifiés du modèle sont remplacés par des variables de travail pris dans les ensembles V_t et V_ω. De même, toute sorte inconnue

[47] Cf. définition A.11.

(par exemple, la sorte du terme "*nil*") est remplacée par une variable de travail avant le début du calcul.

Remarquons qu'il importe de ne pas confondre ces variables de travail avec les paramètres formels introduits par un modèle exigé. En effet, ces paramètres, qui sont *connus* dans l'environnement générique, constituent des expressions fermées. Ainsi, sous le modèle exigé $Ftype[t\ /\ ]$, l'expression de sorte $Seq[t]$ est fermée. Par contre, le terme *nil* est ambigu parce que son type est représenté par une sorte de la forme $Seq[x_t]$ où x_t est une sorte variable qui n'a pas reçu de valeur.

Nous avons vu également au §3.1.1 que nous avions besoin de modèles génériques déclarés au préalable pour le calcul d'une instanciation implicite. En LPG et en FP2, le fait qu'une instanciation soit implicite ou explicite est déterminé par des critères syntaxiques que l'on peut aisément reconstituer à l'aide des exemples donnés au §3.1. Nous intéressant au calcul proprement dit, nous allons à présent adopter la définition suivante, qui rattache cette notion à l'utilisation éventuelle de modèles génériques préalablement connus.

Définition 3.6 *Une instanciation est :*

- **implicite** *si elle est calculée à partir d'un modèle incomplet et d'une* base[48] *(ensemble) de modèles génériques ;*
- **explicite** *si son calcul n'utilise pas de base de modèles génériques.*

La notion de couple d'instanciation vue au §1.2.6 inclut le traitement des déclarations de modèles. Ainsi, les déclarations des modèles $EQNAT$ et $EQSEQ$ induisent les couples d'instanciation suivants :

$$\langle \diamond, Equality[Nat\ /\ =_N] \rangle$$
$$\langle Equality[t_0\ /\ eq_0], Equality[Seq[t_0]\ /\ =_S .Equality[t_0\ /\ eq_0]] \rangle$$

Rappelons que d'après la convention donnée au §3.1.4, nous considérons une *unique* solution comme *la* solution cherchée. Nous traduisons qu'une valeur cherchée p est l'unique élément d'un singleton Q par la notation :

$$p = single(Q)$$

Si Q est vide ou possède plus de deux éléments, c'est un cas d'erreur. De même, définissons, pour une expression e :

$$ground(e) = \begin{cases} e & \text{si } e \text{ est fermée,} \\ \text{erreur} & \text{sinon.} \end{cases}$$

Nous sommes maintenant en possession de tous les outils nécesaires aux calculs.

3.5 Règles d'analyse

L'analyse sémantique s'effectue en deux phases qui correspondent aux traitements de la surcharge de la généricité.

[48]Nous parlons de *base*, car elle joue le rôle de *base de connaissances*.

3.5.1 Prise en compte de la surcharge

Nous donnons ci-après la fonction *unique-term*, qui renvoie la solution de l'analyse sémantique d'un terme, si elle existe et est unique. Elle fait appel à la fonction *solve-term* qui calcule toutes les analyses sémantiques possibles. La reconstruction d'un terme dans lequel ont été levées toutes les ambiguïtés dues à la surcharge est effectuée au moyen de la fonction *make-new-term-with*. Les autres notations utilisées sont :

ω	désigne un opérateur générique quelconque ;
M	est une expression de modèle ;
s'	est une expression de sorte générique ;
$ranks_i(\omega)$	désigne l'ensemble (éventuellement vide) de toutes les déclarations de profils d'arité i, pour l'opérateur surchargé ω ;
solve-clash	prend comme argument une liste de substitutions, cherche à résoudre les conflits de variables, et retourne : en cas de succès : la substitution résultante, en cas d'échec : la valeur "*fail*" ;
$fail(\sigma)$	est vrai si et seulement si $\sigma = fail$;
$t, t_1, \ldots, t_n$	sont des termes.

$$unique\text{-}term(t) = ground(single(solve\text{-}term(t)))$$

$$solve\text{-}term(\omega.M : s') = \bigcup_{\substack{(\omega' : - \to s) \in ranks_0(\omega) \\ s \overset{\varpi}{\approx} s' \\ M \overset{\sigma}{\approx} prim(\pi(\omega' : - \to s))}} \begin{array}{l} \textbf{let } \overline{\sigma} :: solve\text{-}clash(\varpi, \sigma) \\ \quad \textbf{in if } fail(\overline{\sigma}) \textbf{ then } \emptyset \textbf{ else } \{(\omega'.\overline{\sigma}(M) : \overline{\sigma}(s), \overline{\sigma})\} \\ \quad\quad \textbf{endif} \\ \textbf{endlet} \end{array}$$

$$solve\text{-}term(\omega.M(t_1, \ldots, t_n) : s') = \bigcup_{\substack{(\omega' : s_1 \times \cdots \times s_n \to s) \in ranks_n(\omega) \\ s \overset{\varpi}{\approx} s' \\ M \overset{\sigma}{\approx} prim(\pi(\omega' : s_1 \times \cdots \times s_n \to s))}} \begin{array}{l} \textbf{let } make\text{-}new\text{-}term\text{-}with((u'_1, \overline{\sigma}'_1), \ldots, (u'_n, \overline{\sigma}'_n)) :: \\ \quad \textbf{let } \overline{\sigma} :: solve\text{-}clash(\varpi, \sigma, \overline{\sigma}'_1, \ldots, \overline{\sigma}'_n) \\ \quad\quad \textbf{in if } fail(\overline{\sigma}) \textbf{ then } \emptyset \\ \quad\quad\quad \textbf{else } \{(\omega'.\overline{\sigma}(M)(\overline{\sigma}(u'_1), \ldots, \overline{\sigma}(u'_n)) : \overline{\sigma}(s), \\ \quad\quad\quad\quad \overline{\sigma})\} \\ \quad\quad \textbf{endif} \\ \quad \textbf{endlet} \\ \textbf{in} \displaystyle\bigcup_{\substack{(u_1, \overline{\sigma}_1) \in solve\text{-}term(t_1, s_1) \\ \vdots \\ (u_n, \overline{\sigma}_n) \in solve\text{-}term(t_n, s_n)}} make\text{-}new\text{-}term\text{-}with((u_1, \overline{\sigma}_1), \ldots, (u_n, \overline{\sigma}_n)) \\ \textbf{endlet} \end{array}$$

On remarque que l'analyse sémantique opère par résolution d'équations entre expressions de sortes, expressions données d'une part par les divers profils d'un opérateur surchargé, d'autre part, par les analyses sémantiques possibles pour les arguments.

3.5.2 Calcul d'instanciation implicite

Soient RM le modèle exigé courant, et M un modèle incomplet. S'il est possible de déduire de M un couple d'instanciation en utilisant la base $\mathcal{B}$, il est reconstitué par la fonction *implicit-instantiation*, la fonction *couples* calculant tous les couples d'instanciation qui sont solutions.

$$\begin{array}{rcl} \mathit{implicit\text{-}instantiation}(RM, M) & = & \mathit{ground}(\mathit{single}(\mathit{couples}(RM, \langle M, M \rangle))) \\ \\ \mathit{couples}(RM, \langle M_1, M_2 \rangle) & = & \textbf{if } M_1 \overset{\sigma}{\preceq} RM \textbf{ then } \{\langle \sigma(M_1), \sigma(M_2) \rangle\} \\ & & \textbf{else} \displaystyle\bigcup_{\substack{\langle \mu_1, \mu_2 \rangle \in \mathcal{B} \\ \mu_2 \overset{\sigma'}{\approx} M_1}} \mathit{couples}(RM, \langle \sigma'(\mu_1), \sigma'(M_2) \rangle) \\ & & \textbf{endif} \end{array}$$

Montrons d'abord le *modus operandi* sur un exemple.

Exemple 3.7 *Reprenons "$\leqslant_S$", l'opérateur d'ordre lexicographique des séquences (cf. exemple 1.25 ou annexe, §B.6), et soit l'expression :*

$$\leqslant_S([[0]], [[1]]) \tag{3.6}$$

Nous supposons que la base $\mathcal{B}$ contient les couples d'instanciation suivants :

$$\langle \diamond, \overbrace{\mathit{Total\text{-}Order}[\mathit{Nat} \mathbin{/} \leqslant_N, =_N]}^{(I)} \rangle$$

$$\langle \mathit{Total\text{-}Order}[t \mathbin{/} \mathit{rel}, \mathit{eq}], \overbrace{\mathit{Total\text{-}Order}[\mathit{Seq}[t] \mathbin{/} \leqslant_S.\mathit{Total\text{-}Order}[t \mathbin{/} \mathit{rel}, \mathit{eq}], =_S.\mathit{Equality}[t \mathbin{/} \mathit{eq}]]}^{(II)} \rangle$$

L'analyse des arguments nous fournit le modèle incomplet :

$$\mathit{Total\text{-}Order}[\mathit{Seq}[\mathit{Nat}] \mathbin{/} \mathit{rel}_0, \mathit{eq}_0]$$

où rel_0 et eq_0 sont des variables de travail. Le calcul du couple d'instanciation de "$\leqslant_S$" dans l'expression (3.6) s'effectue alors comme suit — rel_1 et eq_1 sont également des variables de travail — :

$$\mathit{couples}(\diamond, \langle \overbrace{\mathit{Total\text{-}Order}[\mathit{Seq}[\mathit{Nat}] \mathbin{/} \mathit{rel}_0, \mathit{eq}_0]}^{(1)}, \mathit{Total\text{-}Order}[\mathit{Seq}[\mathit{Nat}] \mathbin{/} \mathit{rel}_0, \mathit{eq}_0] \rangle) \tag{3.7}$$

$$\begin{aligned} = \; & \mathit{couples}(\diamond, \langle \overbrace{\mathit{Total\text{-}Order}[\mathit{Nat} \mathbin{/} \mathit{rel}_1, \mathit{eq}_1]}^{(2)}, \\ & \quad \mathit{Total\text{-}Order}[\mathit{Seq}[\mathit{Nat}] \mathbin{/} \leqslant_S.\mathit{Total\text{-}Order}[\mathit{Nat} \mathbin{/} \mathit{rel}_1, \mathit{eq}_1], =_S.\mathit{Equality}[\mathit{Nat} \mathbin{/} \mathit{eq}_1]] \rangle) \end{aligned} \tag{3.8}$$

$$\begin{aligned} = \; & \mathit{couples}(\diamond, \langle \diamond, \\ & \quad \mathit{Total\text{-}Order}[\mathit{Seq}[\mathit{Nat}] \mathbin{/} \leqslant_S.\mathit{Total\text{-}Order}[\mathit{Nat} \mathbin{/} \leqslant_N, =_N], =_S.\mathit{Equality}[\mathit{Nat} \mathbin{/} =_N]] \rangle) \end{aligned} \tag{3.9}$$

$$= \{\langle \diamond, \mathit{Total\text{-}Order}[\mathit{Seq}[\mathit{Nat}] \mathbin{/} \leqslant_S.\mathit{Total\text{-}Order}[\mathit{Nat} \mathbin{/} \leqslant_N, =_N], =_S.\mathit{Equality}[\mathit{Nat} \mathbin{/} =_N]] \rangle\} \tag{3.10}$$

d'où :

$$\begin{aligned}
&\textit{implicit-instantiation}(\diamond, \textit{Total-Order}[\textit{Seq}[\textit{Nat}] \;/\; \textit{rel}_0, \textit{eq}_0]) \\
&\quad = \textit{ground}(\textit{single}(\textit{couples}(\diamond, \langle \textit{Total-Order}[\textit{Seq}[\textit{Nat}] \;/\; \textit{rel}_0, \textit{eq}_0], \textit{Total-Order}[\textit{Seq}[\textit{Nat}] \;/\; \textit{rel}_0, \textit{eq}_0]\rangle))) \\
&\quad = \langle \diamond, \textit{Total-Order}[\textit{Seq}[\textit{Nat}] \;/\; \leqslant_S.\textit{Total-Order}[\textit{Nat} \;/\; \leqslant_N, =_N], =_S.\textit{Equality}[\textit{Nat} \;/\; =_N]]\rangle
\end{aligned}$$

(3.7) $\Longrightarrow$ (3.8) par unification de (1) avec (II), (l'unification de (1) avec (I) échoue).

(3.8) $\Longrightarrow$ (3.9) par unification de (2) avec (I).

(3.9) $\Longrightarrow$ (3.10) parce que tout modèle inclut "$\diamond$".

L'instance complète, qui est unique, est donc :

$$\leqslant_S.\textit{Total-Order}[\textit{Seq}[\textit{Nat}] \;/\; \leqslant_S.\textit{Total-Order}[\textit{Nat} \;/\; \leqslant_N, =_N], =_S.\textit{Equality}[\textit{Nat} \;/\; =_N]]$$

3.5.3 Critères et preuves

La règle donnée plus haut calcule *toutes* les instanciations implicites d'un type ou d'un opérateur. Lorsque le résultat est un singleton, il contient l'instanciation qui va être utilisée. Deux preuves sont à donner pour le calcul des instanciations implicites :

- preuve de **correction** : toutes les instanciations obtenues sont correctes ;
- preuve de **complétude** : nous obtenons toutes les instanciations déductibles de la base.

Comme nous l'avons fait déjà remarquer, le cas particulier utilisé en pratique est celui où :

$$\textit{couples}(RM, \langle M_1, M_2\rangle)$$

est réduit à un unique élément. *Ce n'est qu'un cas particulier :* étant donné que l'utilisateur peut écrire des expressions ambiguës, les critères que nous venons de dégager se situent nécessairement au niveau du calcul de *toutes* les réponses possibles.

3.5.3.1 Correction

Considérons le calcul de *couples*$(RM, \langle M_1, M_2\rangle)$. Par définition de la sémantique des expressions de modèles (cf. §3.3.3), M_2 représente des instanciations possibles d'une propriété exigée.

- $\boxed{M_1 \overset{\sigma}{\preceq} RM}$

  Si les expressions $\sigma(M_1)$ et $\sigma(M_2)$ sont fermées, alors, d'après les conventions du §1.2.6 :

  $$\langle \sigma(M_1), \sigma(M_2)\rangle$$

  est un couple d'instanciation correct[49]. $\square$

- $\boxed{M_1 \not\preceq RM}$

  S'il existe $\langle \mu_1, \mu_2\rangle \in \mathcal{B}$, tel que $\mu_2 \overset{\sigma'}{\approx} M_1$, alors, compte tenu de la sémantique de la substitution σ', $\sigma'(M_2)$ représente également des instanciations possibles de la même propriété exigée. $\square$

[49] Au niveau des signatures : rappelons que nous ne nous intéressons pas à la validité des équations d'une propriété.

3.5.3.2 Complétude

La complétude du calcul de $couples(RM, \langle M_1, M_2 \rangle)$ s'obtient par construction, dès lors que pour tout modèle générique $\langle \mu_1, \mu_2 \rangle$ de la base, tel que :

$$\mu_2 \overset{\sigma'}{\approx} M_1$$

nous reportons le résultat par σ' et tentons par la suite du calcul (appel récursif de la fonction *couples*) de satisfaire les conditions données par l'expression de modèle $\sigma'(\mu_1)$. □

3.5.4 Terminaison

Montrons sur un exemple que la terminaison du calcul d'une instanciation implicite n'est pas nécessairement assurée.

Exemple 3.8 *Composition d'un opérateur avec lui-même*[50].

```
prop For-Twice[t / f]
  opns f : t → t
endprop

enr req For-Twice[t / f]
   opns  twice : t → t
   vars  x     : t
   rules
      <>  twice(x)  ==>  f(f(x))
endenr
```

```
model TWICE-NAT is
   For-Twice[Nat / succ]
endmodel

model TWICE-TWICE req
   For-Twice[t / f] is
   For-Twice[t / twice]
endmodel
```

Considérons l'expression *twice*(0) et supposons que *TWICE-NAT* et *TWICE-TWICE* sont dans la base utilisée. On peut voir sans appliquer les règles précédentes que le calcul de l'instanciation implicite diverge. En effet, toutes les instanciations possibles sont :

TWICE-NAT
TWICE-TWICE.TWICE-NAT
TWICE-TWICE.TWICE-TWICE.TWICE-NAT
$\ddots$

Notre calcul étant complet, il ne peut pas terminer dès lors qu'il tente de retourner toutes les instanciations possibles. Si nous enlevons *TWICE-NAT* de la base, alors le calcul boucle, en essayant d'évaluer :

TWICE-TWICE.TWICE-TWICE.TWICE-TWICE...

La non terminaison provient de ce qu'une sorte formelle est présent dans une racine dans le modèle *TWICE-TWICE*.

Théorème 3.9 *Si la base utilisée est finie, et s'il n'y existe pas de modèle générique qui admet une sorte formelle à une racine, alors tout calcul d'une instanciation implicite termine.*

[50] Contre-exemple dû à Philippe Schnoebelen.

Preuve d'après la profondeur des expressions de modèles, en définissant celle-ci de façon analogue à la profondeur des termes (cf. §0.1.1.2).

Dans tout couple d'instanciation $\langle \mu_1, \mu_2 \rangle$ provenant d'une déclaration de modèle éventuellement générique, μ_1 est soit "$\diamond$", soit obtenu par renommage bijectif de la signature ordonnée d'une propriété exigée p. Dans ce dernier cas, il est de la forme :

$$p[t_1, \ldots, t_u \;/\; f_1, \ldots, f_v]$$

où $t_1, \ldots, t_u$ sont des sortes formelles, et $f_1, \ldots, f_v$ des opérateurs formels. Par hypothèse, aucune sorte formelle n'est présente à une racine des expressions de sortes de μ_2.

Soit à présent un couple $\langle M_1, M_2 \rangle$ dont on cherche à calculer :

$$couples(RM, \langle M_1, M_2 \rangle)$$

avec $M_1 \not\lesssim RM$. Supposons qu'il existe $\langle \mu_1, \mu_2 \rangle \in \mathcal{B}$, tel que $\mu_2 \overset{\sigma'}{\approx} M_1$, ce qui provoque l'appel :

$$couples(RM, \langle \sigma'(\mu_1), \sigma'(M_2) \rangle)$$

Les sortes variables de travail de $\sigma'(\mu_1)$ sont situées à une occurrence strictement moins profonde que les sortes variables de travail de M_1. Ce sont les sortes qui dirigent le calcul puisqu'elles correspondent aux informations fournies de façon partielle par l'utilisateur ou aux résultats de la phase d'analyse sémantique. Par conséquent, la fonction *couples* termine. En cas de succès, ses appels récursifs mènent nécessairement à une unification avec un couple d'instanciation dont le premier membre est soit "$\diamond$", soit inclus dans le modèle exigé RM. $\square$.

Note Une possibilité serait de refuser la déclaration de modèles génériques tels que *TWICE-TWICE* — c'est le choix de LPG —, ce qui reviendrait à se priver de la possibilité de les instancier, alors que par exemple, l'expression :

$$twice.TWICE\text{-}TWICE.TWICE\text{-}NAT(0) ==> 4$$

ne pose aucun problème. La règle suivie par FP2 est que les modèles génériques avec une sorte formelle à une racine sont acceptés et peuvent être eux-mêmes instanciés dans une expression, mais ils ne sont pas utilisés dans le calcul d'une instanciation implicite. Ainsi, *twice*(0) est analysé comme *twice.TWICE-NAT*(0). De même, l'expression *twice.TWICE-TWICE*(0) est comprise comme étant *twice.TWICE-TWICE.TWICE-NAT*(0).

3.6 Raccourcis de notation en OBJ3 — Comparaison

OBJ3 possède lui aussi un certain nombre de règles qui permettent à l'utilisateur de ne pas indiquer la totalité des liaisons d'un morphisme de présentations. Nous introduisons la notion de sorte principale, et donnons brièvement ces règles, extraites de [72].

Définition 3.10 ([72]) *La* **sorte principale** *d'un module est définie comme :*

- *la première sorte nouvelle introduite dans le module, si cette sorte existe,*
- *sinon la sorte principale de la première théorie paramètre, si cette sorte existe,*
- *sinon la première sorte principale des modules importés.*

Proposition 3.11 ([72]) *Chaque module admet une unique sorte principale.*

Il est possible d'omettre la mention de certaines liaisons du morphisme, suivant les règles ci-après, données dans l'ordre de leur application :

- une liaison "`s to s'`" peut être omise si `s` et `s'` sont toutes deux des sortes principales,
- toutes les liaisons de la forme "`s to s`" ou "`op to op`" peuvent être omises,
- une liaison "`op to op'`" peut être omise si elle est telle que, dans l'objet image de la vue, il n'existe qu'un seul `op''` dont le profil est égal à l'image du profil de `op`.

Joseph Goguen montre dans [72] comment est reconstruite de façon univoque une vue à l'aide de l'une de ses abréviations. Il s'ensuit que si une vue θ est une abréviation de deux vues φ et φ', alors $\varphi = \varphi'$.

Une vue vers un objet dont la mention d'aucun couple n'est nécessaire est une **vue par défaut** (*default view*) : dans ce cas seule suffit la mention de l'objet image du morphisme.

Exemple 3.12 *Vues par défaut : arbres binaires d'entiers suivant l'ordre croissant (cf. exemple 1.37), ajout au type Nat d'un élément minimum et d'un élément maximum (cf. exemple 1.38).*

```
obj OTINCNAT is protecting OTREE[NAT] .
jbo

obj LCNAT is extending LATTICE[NAT] .
jbo
```

Les morphismes utilisés sont respectivement :

$$\begin{array}{ll} \texttt{TOSET} \to \texttt{NAT} & [\texttt{Elt} \mapsto \texttt{Nat}, < \mapsto <] \\ \texttt{POSET} \to \texttt{NAT} & [\texttt{Elt} \mapsto \texttt{Nat}, < \mapsto <] \end{array}$$

Les facilités d'instanciation de FP2 (ou de LPG) sont assez mnémotechniques et relativement faciles à mettre en application au niveau de la programmation. Leur défaut est de dépendre du contexte : il suffit d'ajouter une déclaration de modèle et une expression qui était complètement univoque peut fort bien ne plus l'être, ce qui représente un défaut important si l'on se place dans une optique de réutilisabilité des spécifications. Les règles d'abréviations dans les vues d'OBJ3 sont certainement moins immédiates, mais ne sont pas perturbées par l'ajout de nouveaux modules.

Partie II

Chapitre 4

Exceptions

Nous allons aborder à présent un point important qui est celui des fonctions partielles. En effet, il est essentiel de pouvoir spécifier qu'un opérateur est partiellement défini, comme il est nécessaire de prévoir le comportement de FP2 lorsqu'une erreur due à une mauvaise utilisation d'un opérateur partiel survient dans une évaluation, tant dans une "simple évaluation fonctionnelle" que durant l'exécution d'un processus. Nous ne traiterons ici les exceptions que dans un cadre algébrique, les lecteurs intéressés par des définitions précises des diverses terminologies employées au niveau logiciel (exceptions, défaillances, erreurs) pourront se reporter à [49]. Signalons également que l'axiomatique des programmes avec exceptions est étudiée dans [46, 47, 48, 178].

Nous allons d'abord rappeler que si nous considérons les opérateurs d'exception comme des constructeurs d'une sorte existant déjà, cette sorte n'est plus protégée. Ensuite, nous explicitons complètement la solution qui a été retenue dans LPG et FP2. Nous en donnons une description opérationnelle, puis nous montrons que, sous certaines conditions vérifiées en FP2, notre introduction des exceptions ne modifie pas la propriété de terminaison (ou de non terminaison) d'un système de réécriture. Nous poursuivons le chapitre par une comparaison avec les autres concepts qui ont été proposés pour le traitement des erreurs, en particulier les sortes ordonnées. Ensuite, la sémantique algébrique que nous proposons fait apparaître que les exceptions s'inscrivent complètement dans le cadre des algèbres avec sortes ordonnées. Outre qu'il assure l'existence d'une algèbre initiale, ce résultat permet de traiter les exceptions avec paramètres génériques, ce qui n'existait pas jusqu'à présent. Nous retrouvons les processus de FP2 en proposant une extension naturelle des exceptions — déjà esquissée en [159] — en vue d'un traitement local dans un processus. Enfin, nous rappelons comment ce problème est résolu en Lisp et traitons de la compilation des exceptions de FP2.

4.1 Première approche des cas d'erreur

Considérons, dans le cadre des entiers naturels, la spécification suivante[51] ("+" est, bien sûr, supposé défini) :

$$\begin{aligned} 0 * p &== 0 \\ succ(n) * p &== (n * p) + p \\ n - 0 &== n \\ 0 - succ(p) &== nat\text{-}error \end{aligned} \tag{4.1}$$

[51] Des exemples analogues sont donnés dans [15, 22, 10].

$$succ(n) - succ(p) \quad == \quad n - p$$

Si nous considérons *nat-error* comme un opérateur de codomaine *Nat*, nous avons alors ajouté au type *Nat* une nouvelle constante, en fait un nouveau constructeur, puisque nous désirons différencier cet opérateur des naturels "de départ", "0" et "*succ*(...)". Nous complétons donc la définition des opérateurs déjà connus, la solution qui vient à l'esprit étant de propager *nat-error*. Nous ajoutons les équations :

$$succ(nat\text{-}error) \quad == \quad nat\text{-}error \tag{4.2}$$
$$nat\text{-}error * p \quad == \quad nat\text{-}error$$
$$n * nat\text{-}error \quad == \quad nat\text{-}error \tag{4.3}$$

nat-error étant donc un terme de sorte *Nat* :

$$\begin{aligned} 0 * nat\text{-}error &==_{Nat} 0 && \text{(d'après (4.1))} \\ &==_{Nat} nat\text{-}error && \text{(d'après (4.3))} \end{aligned}$$

Il vient donc : $0 ==_{Nat} nat\text{-}error$.

Et, d'après (4.2) :

$$\begin{aligned} succ(0) &==_{Nat} nat\text{-}error \\ succ(succ(0)) &==_{Nat} nat\text{-}error \\ &\vdots \end{aligned}$$

Tous les éléments de sorte *Nat* sont équivalents à *nat-error*. Le seul modèle de notre spécification est l'algèbre triviale[52], ceci parce que nous avons pu appliquer deux équations au terme $0 * nat\text{-}error$: une équation *définissant* un opérateur (en l'occurrence "*"), et une équation de *propagation*. Pour éviter pareil cas de figure, il nous faut traiter différemment les opérateurs d'exception et les autres.

4.2 Description opérationnelle des exceptions

Nous allons poser l'existence d'un ensemble Φ regroupant tous les **opérateurs d'exception**. Nous considérerons de plus que nous pouvons créer autant d'opérateurs d'exception que nous le voulons. Par convention, ces opérateurs seront signalés par un "!" ("*!zero-divide*", "*!undefined-value*").

Les exceptions de FP2 n'admettent pas de paramètre[53]. Toutefois, étant donné que la majeure partie de ce chapitre se généralise facilement aux exceptions avec paramètres, éventuellement génériques (présentes en LPG), nous prévoyons les paramètres dans nos définitions. Ceci revient à distinguer

[52] Étant donné une signature $\Sigma = (S, \Omega)$, l'algèbre **triviale** Z sur Σ est définie par :

$$\begin{aligned} (\forall s \in S), && Z_s &= \{s\} \\ (\forall \omega \in \Omega_{__\to s}), && Z_\omega &= s \\ (\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}), && Z_\omega(s_1, \ldots, s_n) &= s \end{aligned}$$

C'est une algèbre terminale dans la catégorie $\mathbf{ALG}_\Sigma$.

[53] Leur déclaration est par conséquent *facultative*.

d'une part les opérateurs d'exception, d'autre part les **termes exceptionnels**, qui sont des termes dont la racine est un opérateur d'exception.

Toute fonction peut dès lors rendre un résultat appartenant à son codomaine ou un résultat *exceptionnel*. Nous verrons dans l'étude sémantique que nous traduirons ce second comportement en *indiçant* l'opérateur d'exception par la sorte attendue. L'utilisateur n'a jamais à indiquer cet indice, qui est entièrement à la charge du compilateur. Aussi, nous le laissons implicite dans les exemples et la description opérationnelle.

On appelle **déclenchement d'exception** l'occurrence d'un terme exceptionnel durant une évaluation et **récupération d'exception** le passage d'un terme exceptionnel à un terme qui lui est différent, exceptionnel ou non[54]. Nous allons expliquer ci-après comment sont organisées les interactions entre termes exceptionnels et autres termes.

4.2.1 Règles de précondition

Opérationnellement, une **règle de précondition** s'apparente à une règle de réécriture conditionnelle, mais la partie droite est un terme formé à l'aide d'un opérateur d'exception.

$$f(x_1, \ldots, x_n) \mid g(\ldots) ==> !\mathit{exception\text{-}op}(\ldots)$$

$g(\ldots)$ est une **précondition** de f : les valeurs $x_1, \ldots, x_n$ pour lesquelles $g(\ldots)$ est vrai *déclenchent* l'exception "*!exception-op*", et dans le cas contraire, nous dirons que la précondition *échoue*. N'importe quel opérateur, même constructeur, peut se voir imposer des préconditions. Si toutes les préconditions de f échouent, les arguments appartiennent bel et bien au domaine de définition de f : dans le cas où f n'est pas constructeur, ce n'est qu'après l'échec de toutes les préconditions que l'on cherche à appliquer une règle de réécriture définissant f.

Sémantiquement, une règle de précondition d'un opérateur f de domaine $s_1 \times \cdots \times s_n$ simule, dans tout modèle A, une rétraction[55] de $A_{s_1} \times \cdots \times A_{s_n}$ vers un de ses sous-ensembles. Le **domaine de définition** de f (noté déf(f)) est l'image de la composition de toutes les règles de préconditions de f.

Exemple 4.1 *Le constructeur du type des rationnels positifs admet une précondition :*

```
type PRat      -- Rationnels positifs.
  cons  //       : Nat × Nat
  opns  canonic  : PRat        →  PRat
  vars  i, j, k  : Nat
  prcds
     <>  i // 0  |  true  ==>  !zero-divide
  rules
     <>  canonic(i // j)  ==>  let  k :: i gcd j
                                    in (i div k) // (j div k)
                               endlet
endtype
```

[54] Certains auteurs considèrent que la récupération d'erreur est un mécanisme qui permet de passer d'un terme exceptionnel à un terme non exceptionnel. Nous considérons le fait que la récupération permet de remplacer un terme exceptionnel par le résultat de l'évaluation d'un autre terme, évaluation qui peut fort bien déclencher à nouveau une exception.

[55] Cf. définition A.11 (annexe A).

4.2.2 Déclenchement d'exception dans une évaluation

En l'absence de récupération, si un déclenchement d'exception se produit lors de l'évaluation d'un terme, le terme tout entier est *remplacé* par l'évaluation du terme exceptionnel : il s'agit d'un mécanisme d'*absorption*. En particulier, si le déclenchement se produit lors du calcul d'un argument d'un opérateur, l'évaluation des arguments restant à calculer est *abandonnée*, et le résultat est le terme exceptionnel. Nous considérerons que les arguments d'un opérateur sont évalués de gauche à droite ; cette convention va à l'encontre d'un principe de la programmation fonctionnelle selon lequel l'ordre d'évaluation des arguments d'un opérateur est indifférent, mais elle assure le déterminisme.

Nous allons commencer à formaliser ces notions. En accord avec la description précédente, nous dirons qu'un terme est **exceptionnel** si l'un de ses arguments est un terme exceptionnel, ou s'il est construit à l'aide d'un opérateur de Φ :

$$\mathit{is\text{-}exceptional}(\overline{\omega(t_1,\ldots,t_n)}) = ((\exists i : 1 \leqslant i \leqslant n), \mathit{is\text{-}exceptional}(t_i)) \vee (\omega \in \Phi)$$

Nous avons caractérisé les termes exceptionnels : maintenant nous allons définir formellement la **règle d'absorption :**

$$\boxed{\begin{array}{l}(\forall \overline{\omega(t_1,\ldots,t_n)}),\\ \qquad ((\exists i, 1 \leqslant i \leqslant n, \mathit{is\text{-}exceptional}(t_i)) \Longrightarrow \overline{\omega(t_1,\ldots,t_n)} \equiv t_{\min\limits_{1\leqslant j\leqslant n}\{j \mid \mathit{is\text{-}exceptional}(t_j)\}})\end{array}}$$

Nous avons tout simplement indiqué que lors de l'évaluation d'un terme, c'est l'exception "la plus à gauche" qui l'emporte. Comme nous l'avons remarqué plus haut, c'est cette convention qui assure le déterminisme. À remarquer qu'elle évoque aussi une stratégie "par valeur" pour la réécriture des termes contenant des opérateurs d'exception. En effet, soit "$!e$" un opérateur d'exception de domaine *Nat*, alors :

$$!e(1\ \mathit{div}\ 0) \longrightarrow !e(!\mathit{zero\text{-}divide}) \longrightarrow !\mathit{zero\text{-}divide}$$

Remarque Il est possible de convenir que la mention d'exceptions n'est possible qu'en partie droite d'une règle de précondition. Ce choix, très voisin de l'approche au moyen de sous-sortes (cf. §4.4.2), est celui de LPG 1.8, et se justifie parfaitement pour un langage de spécification. Néanmoins, nous permettons en FP2 la mention d'opérateurs d'exception dans les parties droites des règles de réécriture des opérateurs (et dans les postconditions des règles de transition des processus, nous le verrons en §4.6.2). Il nous semble en effet que c'est une optique plus réaliste pour un langage de spécification qui se veut également utilisable pour la programmation d'applications ambitieuses : on permet ainsi à l'utilisateur de mener de front le calcul et la vérification de l'appartenance au domaine de définition (par exemple, un analyseur syntaxique construit l'arbre abstrait au fur et à mesure qu'il vérifie que l'entrée est bien formée).

En fait, pour simplifier le formalisme dans une prochaine version, nous suggérerions la disparition de la distinction entre règles de précondition et règles de réécriture, au profit d'une modification de la sémantique : l'adoption de la réécriture conditionnelle. Nous imposerions alors que la partie droite de toute règle de réécriture conditionnelle appliquée à un constructeur soit un terme exceptionnel.

Pour clore cette remarque, nous signalons que l'algorithme de compilation du filtrage, décrit en [164] et utilisé en FP2 (cf. §2.1.3), se généralise facilement aux règles de réécriture conditionnelles. C'est ce qui a permis de l'utiliser pour compiler les règles de précondition. Quant aux opérateurs constructeurs, ils sont par conséquent compilés comme les opérateurs non constructeurs, et la valeur "par défaut", en cas d'échec de toutes les préconditions, dépend du choix de représentation pour la sorte en cours de compilation (cf. §2.1.2.2 et tableau p. 86).

4.2.3 Récupération d'exception

Un **récupérateur** permet de transformer un terme d'exception en un autre terme. Si un terme de sorte u était attendu en lieu et place de l'exception, le résultat du récupérateur pourra être soit un terme de sorte u, soit un autre terme d'exception. Un récupérateur ne traite *que* certains termes d'exception : il laisse invariants les autres termes d'exception et les termes non exceptionnels. Par conséquent, pour définir un récupérateur, il suffit de le spécifier uniquement dans les cas où il n'équivaut pas à l'identité :

$$\begin{array}{lll} \textbf{when} & !exc_1(\tau_{1,1},\ldots,\tau_{1,\text{arity}(!exc_1)}) & \textbf{then} \;\; t_1 \\ \vdots & & \\ \textbf{when} & !exc_n(\tau_{n,1},\ldots,\tau_{n,\text{arity}(!exc_n)}) & \textbf{then} \;\; t_n \\ \textbf{otherwise} & & \qquad\;\; t_{n+1} \end{array}$$

Les "$\tau_{i,j}$" jouent un rôle de filtres — en fait, on se restreint usuellement à des variables pour chaque "$\tau_{i,j}$" —, "**otherwise**" récupère tous les termes exceptionnels qui ne sont pas construits à l'aide de $!exc_1, \ldots, !exc_n$. Si cette clause est absente, le récupérateur équivaut à l'identité pour les exceptions non mentionnées : c'est le cas de propagation par défaut. Pour mentionner qu'un tel récupérateur s'applique au résultat de l'évaluation d'un terme t_0 :

$$\begin{array}{llll} \textbf{rec} & t_0 & & \\ & \textbf{when} & !exc_1(\tau_{1,1},\ldots,\tau_{1,\text{arity}(!exc_1)}) & \textbf{then} \;\; t_1 \\ & \vdots & & \\ & \textbf{when} & !exc_n(\tau_{n,1},\ldots,\tau_{n,\text{arity}(!exc_n)}) & \textbf{then} \;\; t_n \\ & \textbf{otherwise} & & \qquad\;\; t_{n+1} \\ \textbf{endrec} & & & \end{array}$$

Dans une optique de réécriture, les trois phases de l'évaluation sont :

1. réécriture de t_0,
2. évaluation du récupérateur,
3. réécriture du résultat du récupérateur.

Nous notons $\mathcal{R}$ le système de réécriture utilisé (supposé convergent). On peut comprendre un récupérateur comme étant lui-même un système de réécriture $\mathcal{R}'$ dont on n'applique qu'un pas au plus. Ainsi la formulation complète de l'évaluation de ce terme est :

$$\boxed{((t_0{\downarrow}_{\mathcal{R}}) \xrightarrow{1}_{\mathcal{R}'}){\downarrow}_{\mathcal{R}}}$$

où "$t \xrightarrow{1}_{\mathcal{R}'}$" représente :

⟶ le résultat de l'application à t, suivant le système $\mathcal{R}'$, d'un pas de réécriture seulement, lorsque t est réductible,

⟶ t dans le cas contraire.

Exemple 4.2 *Division par 0 et récupération.*

1 *div* 0 $
  -- ("$" marque la fin d'une expression.)
!*zero-divide*

rec 1 *div* 1
    when !*zero-divide* **then** 0
endrec $
1

rec 1 *div* 0
    when !*zero-divide* **then** 0
endrec $
0

rec 1 *div* 0
    when !*foo-bar* **then** 0
endrec $
!*zero-divide*

4.2.4 Exceptions avec paramètres formels

Une limitation importante de ce *modus operandi* est l'impossibilité de l'implanter sans tests de sorte, dès que l'on introduit des exceptions à paramètres formels. On s'en convaincra par les exemples suivants.

Soit "$!e_0$", un opérateur d'exception défini sous le modèle exigé *Ftype*[*t* /], et dont le domaine est *t*. Supposons que le résultat d'une évaluation soit $!e_0(0)$. En ce qui concerne l'affichage du résultat, il est immédiat d'écrire que ce dernier est un terme exceptionnel préfixé par "$!e_0$", mais si l'on n'a pas conservé pendant l'exécution l'information de sorte pour l'argument de "$!e_0$", alors il est impossible de la retrouver selon le principe de "décompilation" vu au §2.1.4.

D'autre part, un inconvénient beaucoup plus grave est que la non conservation de cette information de sorte peut provoquer des catastrophes lors de la récupération d'exceptions, car il n'existe alors aucun moyen de s'assurer que l'instanciation effectuée lors du déclenchement de l'exception est la même que celle du récupérateur. Par exemple, soit $f : t \rightarrow t$, un opérateur défini sous le modèle exigé *Ftype*[*t* /], et soient x et j deux variables de sortes respectives t et *Nat* :

$f(x)$ ==> **rec** $!e_0(x)$
    when $!e_0(j)$ **then** x
endrec

Cette spécification vérifie les conditions de concordance des sortes et des variables données au § précédent. L'accepter suppose que l'on sache distinguer les deux cas :

$f(0)$	qui doit retourner	0
$f(true)$	...	$!e_0(true)$

la différence n'étant possible que par un test de sorte du paramètre de $!e_0$.

Le vice de l'exemple précédent provient de ce que la partie gauche préfixée par $!e_0$ ne doit pas filtrer tous les sous-termes préfixés par $!e_0$. Pour supprimer cet inconvénient, une suggestion pourrait être de n'admettre que les variables les plus générales dans les parties gauches des récupérateurs, c'est-à-dire de refuser celles qui ne correspondent pas à un renommage des paramètres formels (c'est la convention de LPG). Malheureusement, le même problème se pose, "à l'envers" cette fois. Soit à présent $g : t \rightarrow t$ (x et y étant des variables de sorte t) :

$g(x)$ ==> **rec** $!e_0(0)$
    when $!e_0(y)$ **then** y
endrec

$g(true) ==> 0$!! et non un résultat de sorte *Bool*, ceci parce que le paramètre formel t est instancié différemment pour g — $[t \mapsto Bool]$ — et pour "$!e_0$" — $[t \mapsto Nat]$. Pour éviter pareil accident, l'opération de filtrage doit s'assurer que l'instanciation de "$!e_0$" est la même que l'instanciation de l'opérateur présidant à l'instanciation de la règle de réécriture à l'intérieur de laquelle se trouve le récupérateur.

Note C'est parce que la présence d'exceptions à paramètres formels détruit le caractère fortement typé du langage que les exceptions de FP2 n'admettent pas de paramètre. En fait, ces problèmes n'existent pas pour les exceptions avec domaine complètement instancié (*Nat*, *Bool*, ...), mais nous avons préféré être homogène.

4.2.5 Homogénéisation

Au niveau de la programmation, une facilité importante des exceptions est qu'elles permettent d'homogénéiser les états *anormaux* d'une évaluation. Si l'on convient que les erreurs de l'environnement du système (lecture au-delà de la fin d'un fichier, débordement d'entiers...) retournent des termes exceptionnels, on permet à l'utilisateur la récupération de ces erreurs. Lorsqu'on souhaite qu'elles interrompent purement et simplement le déroulement d'une évaluation, le même effet est obtenu par l'exception non récupérée. Afin de prendre en compte ces erreurs "système", il est nécessaire d'inclure les exceptions correspondantes dans la spécification d'une interface entre le langage à implanter et le langage (système) sous-jacent. Nous voyons que cette extension du concept d'exception peut être particulièrement utile dans un langage fondé sur les types abstraits, pour l'écriture d'une application ambitieuse, par exemple un système d'exploitation.

Pour signaler une autre homogénéisation : lorsque, durant une évaluation, il est impossible de réduire un sous-terme préfixé par un opérateur non constructeur, le résultat est le terme d'exception "!*nothing-matches*". Cette convention, choisie pour des raisons de protection, assure que tout terme irréductible et non exceptionnel peut s'exprimer uniquement à l'aide de constructeurs.

4.2.6 Remarque importante

Lorsqu'on considère des équivalences entre termes ou expressions d'opérateurs, elles ne sont en général valables qu'en l'absence de déclenchements d'exceptions. En particulier, lors de l'application de formules de simplification dans le contexte des transformations de programmes, il n'est possible de *supprimer* des évaluations que si l'on sait qu'elles ne retourneront pas une exception. Par exemple, Saddek Bensalem décrit dans [9] des opérateurs génériques parmi lesquels :

$$\begin{array}{llcl} \uparrow i : & s_1 \times \cdots \times s_n & \rightarrow & s_i \quad (1 \leqslant i \leqslant n) \\ & (x_1, \ldots, x_n) & \mapsto & x_i \\ \\ \psi : & s & \rightarrow & CP^n[s_1, \ldots, s_n] \end{array}$$

— CP^n représentant le produit cartésien de n sortes (cf. annexe, exemple B.10) — tel que :

$$\psi[f_1, \ldots, f_n](x) ==> \langle f_1(x), \ldots, f_n(x) \rangle$$

et donne la règle suivante :

$$\frac{(\forall i, j : 1 \leqslant {}^{i}_{j} \leqslant n), \text{déf}(f_i) = \text{déf}(f_j)}{\uparrow i \circ \psi[f_1, \ldots, f_i, \ldots, f_n] == f_i}$$

La raison d'être de la condition[56] est de prévenir tout déclenchement d'exception dont la propagation rendrait cette règle inapplicable. En effet :

$$\begin{aligned} \uparrow 1(\psi[*, div](\langle 1, 0\rangle)) & \; ==> \; \uparrow 1(\langle 1 * 0, 1 \; div \; 0\rangle) \\ & \; ==> \; !zero\text{-}divide \end{aligned}$$

4.2.7 Réalisations voisines

Les échappements de Lisp seront étudiés plus loin, au §4.7.1. À présent, nous mentionnons rapidement les réalisations de LPG (version 1.8) [19] et de ML [89].

4.2.7.1 Conditions de LPG 1.8

En LPG 1.8 [19], règles de précondition et récupérateurs sont groupés sous une même rubrique : les **conditions**, auxquelles revient toute la gestion des exceptions. Une condition de **déclenchement** a la forme :

left-part **suchthat** $g(\ldots)$ **raises** *exception-op*$(\ldots)$

où *left-part* doit avoir la forme d'une partie gauche d'équation exécutable. Ces conditions équivalent à nos règles de précondition et ont exactement le même comportement.

Une condition de **récupération** est liée à un opérateur (nécessairement non constructeur) :

$$f(u_1, \ldots, u_n) \; \textbf{when} \; exception\text{-}op(\ldots) \begin{cases} ==> & \ldots \\ \textbf{raises} & \ldots \end{cases}$$

Ce choix revient, dans nos conventions, à "factoriser"un récupérateur équivalent à la condition de récupération, ceci pour toutes les règles de réécriture qui définissent f, y compris la règle "trappe" :

$$f(u_1, \ldots, u_n) \longrightarrow !nothing\text{-}matches$$

appliquée en cas d'échec de toutes les autres règles.

4.2.7.2 *Failures* en ML — Utilisation en LCF

Nous nous plaçons un instant dans le cadre du λ-calcul typé et abordons les *failures* de ML [89], qui n'admettent pas de paramètre et se comportent opérationnellement comme les exceptions de FP2.

`failwith 'exc'` : déclenchement de l'exception `exc`

rec t_0							
when $!exc_1$	**then** t_1			t_0	??	['exc₁']	t_1
⋮		s'écrit en ML :		⋮			
when $!exc_n$	**then** t_n				??	['excₙ']	t_n
otherwise	t_{n+1}				?		t_{n+1}
endrec							

[56]Dans [9], cette condition est donnée sous la forme équivalente :

$$(\forall i : 1 \leqslant i \leqslant n), \text{déf}(f_i) \subseteq \bigcap_{1 \leqslant j \leqslant n} \text{déf}(f_j)$$

Remarquons que les exceptions de ML sont assimilées à un type variable (polymorphe) lors de l'inférence de type :

```
# let clash x = failwith 'error' ;;
clash = - : (* -> **)

# clash 0 ;;
evaluation failed error
```

Note ML offre de plus la possibilité de *reprendre* l'exécution d'un récupérateur tant que la sortie de celui-ci s'effectue par une récupération d'exception. Dans ce cadre, signalons que le langage ALEX [29] intègre au λ-calcul des primitives de *signaux* d'exception, avec possibilité d'*abandonner* un calcul après l'activation d'un signal, ou de *reprendre* ce calcul immédiatement après l'endroit de l'activation, ou encore de *ré-évaluer* avec de nouveaux arguments la fonction qui a provoqué le signal. La sémantique opérationnelle de ces primitives est donnée dans [29].

Nous citons les exceptions de ML afin de signaler leur utilisation dans LCF[57] (Logic for Computable Functions), outil d'aide à la démonstration de théorèmes. Sans entrer dans les détails, mentionnons qu'une *preuve* de LCF est une fonction dont le type[58] est :

$$validation = (theorem\text{-}list \rightarrow theorem)$$

Les preuves à effectuer sont guidées par un *but* :

$$goal = (formula\text{-}list \times formula)$$

Atteindre le but (Γ, f), c'est prouver la formule f sous les hypothèses Γ. Une méthode pour atteindre un but consiste à le décomposer en sous-buts, puis à composer les fonctions de preuve des sous-buts. Ceci s'effectue à l'aide de *tactiques*, qui sont des fonctions d'ordre supérieur.

$$tactic = (goal \rightarrow (goal\text{-}list \times validation))$$

Soient γ un but, $\gamma_1, \ldots, \gamma_n$ des sous-buts, et p une fonction de preuve. La tactique :

$$T(\gamma) = ([\gamma_1, \ldots, \gamma_n], p)$$

est *valide* si pour tous théorèmes $\theta_1, \ldots, \theta_n$ prouvant respectivement les sous-buts $\gamma_1, \ldots, \gamma_n$, alors $p([\theta_1, \ldots, \theta_n])$ est un théorème prouvant γ.

L'*échec* d'une tactique sur un but est représenté par un déclenchement d'exception. L'endroit où cette exception est récupérée donne l'étendue de la partie invalidée de la preuve.

4.3 Exceptions et propriété de terminaison

Théorème 4.3 *Lorsque les équations entre constructeurs peuvent être orientées en un système de réécriture, l'introduction de la règle d'absorption dans un système de réécriture, en l'absence de récupération d'exceptions, ne modifie pas la propriété de terminaison ou de non terminaison de ce système.*

[57] Historiquement, ML (Metalanguage) fut conçu comme support de LCF (cf. [89]). Ce n'est que par la suite que ses qualités intrinsèques le firent connaître pour lui-même.

[58] Rappelons que dans le λ-calcul typé, existe le *type* fonction.

Preuve Nous allons d'abord montrer que le théorème est vrai pour les systèmes de réécriture linéaires à gauche[59]. La seconde partie de la preuve consiste à montrer que, moyennant certaines conventions qui s'appliquent à notre cas[60], nous pouvons ramener tout système de réécriture à un système de réécriture linéaire à gauche. Afin de ne pas créer de digression qui ralentirait l'exposé de ce chapitre, les définitions, lemmes et preuves techniques de cette seconde partie ont été regroupés à l'annexe C.

Posons donc, pour l'instant, les hypothèses suivantes :

- le système de réécriture $\mathcal{R}$ qui définit les opérateurs est linéaire à gauche,
- les opérateurs d'exception sont en nombre fini et n'admettent pas d'arguments.

Une variable ne pouvant pas filtrer un terme exceptionnel, les termes dans lesquels apparaissent des exceptions sont dès lors irréductibles. Ainsi, "(3 + !*zero-divide*) + 2" est irréductible (mais non "!*zero-divide* + (3 + 2)", qui se réduit à "!*zero-divide* + 5").

Construisons pour tous les opérateurs de la signature ayant une arité non nulle, le système de réécriture $\mathcal{S}$ équivalent à la règle d'absorption, soit pour un opérateur f, des règles de la forme :

$$f(x_1, \ldots, x_{i-1}, !exc^1_{s_i}, x_{i+1}, \ldots, x_n) \;==>\; !exc^1_s$$
$$f(x_1, \ldots, x_{i-1}, !exc^1_{s_i}, !exc^2_{s_{i+1}}, \ldots, x_n) \;==>\; !exc^1_s$$

Ceci revient à supposer qu'un tel opérateur, au lieu d'admettre un profil de la forme $s_1 \times \cdots \times s_n \rightarrow s$, est défini sur $(s_1 \cup exception_{s_1}) \times \cdots \times (s_n \cup exception_{s_n}) \rightarrow (s \cup exception_s)$, en considérant que $exception_{s_i}$ représente la sorte qui qualifie les termes exceptionnels employés en lieu et place de termes de sorte s_i. Pour ce qui est du système $\mathcal{S}$, il a un grand nombre de règles, mais est néanmoins fini. On peut constater facilement qu'il est linéaire à droite[61], qu'il est nœthérien (pour toute règle, la profondeur des termes décroît strictement entre la partie gauche et la partie droite), et qu'il n'y a aucune superposition[62] possible entre les parties gauches de $\mathcal{R}$ et les parties droites de $\mathcal{S}$. Rappelons à présent le théorème suivant :

Théorème 4.4 (Bachmair & Dershowitz, 1986 [3]) *Soient $\mathcal{R}$ et $\mathcal{S}$ deux systèmes de réécriture définis sur la même algèbre de termes, tels que $\mathcal{R}$ est linéaire à gauche, $\mathcal{S}$ linéaire à droite, et tels qu'aucune superposition n'est possible entre les parties gauches de $\mathcal{R}$ et les parties droites de $\mathcal{S}$. Alors $\mathcal{R} \cup \mathcal{S}$ est nœthérien si et seulement si $\mathcal{R}$ et $\mathcal{S}$ sont nœthériens.*

Par conséquent, dans notre cas :

$$\boxed{\mathcal{R} \text{ nœthérien} \Longleftrightarrow \mathcal{R} \cup \mathcal{S} \text{ nœthérien}}$$

Nous venons donc de prouver que l'introduction de la règle d'absorption ne modifie pas la propriété de terminaison ou de non terminaison d'un système de réécriture linéaire à gauche. Il est possible d'étendre ce résultat aux exceptions avec paramètres en ajoutant dans $\mathcal{S}$ des "règles d'absorption" analogues pour les opérateurs d'exception admettant des paramètres.

---

[59] C'est-à-dire dont toutes les parties gauches sont linéaires.

[60] Notons que FP2 permet les parties gauches non linéaires (ainsi que les préconditions non linéaires dans les règles des processus).

[61] C'est-à-dire que toutes les parties droites sont linéaires.

[62] Superposition au sens de [3] : deux termes sont superposables si l'un d'eux peut être unifié avec un sous-terme de l'autre, sous-terme non réduit à une variable.

Ce résultat constitue donc la preuve d'une conjecture de [150]. Dans sa thèse [150], Jean-Luc Rémy avait démontré ce résultat lorsque le système de réécriture $\mathcal{R}$ pouvait être orienté par l'ordre de Kamin et Lévy[63].

4.4 Comparaison avec d'autres concepts

Nous allons à présent passer en revue les principales propositions de traitements d'erreur. Ce tour d'horizon nous permettra de mieux situer nos choix pour l'étude sémantique.

4.4.1 Algèbres partielles

La théorie la plus séduisante du point de vue mathématique est sans nul doute celle des **algèbres partielles**. Les notions de base sont données dans [90], et des résultats puissants ont été dégagés dans le cadre du projet CIP (Computer-aided Intuition-guided Programming) [30, 31]. Sémantiquement, cette approche est tout à fait satisfaisante et complète (existence d'une algèbre initiale, définitions hiérarchiques [30]), mais dès lors que les fonctions sont partielles, le comportement des cas de non définition est ignoré, et, en particulier, la notion de récupération d'exceptions est totalement inexistante. Ajoutons que ces auteurs considèrent des prédicats exprimant qu'un terme est défini, et introduisent une règle qui traduit que tout sous-terme d'un terme défini est défini. On peut la comprendre comme une règle "duale" de notre règle d'absorption.

4.4.2 Sous-sortes

L'approche au moyen de sortes et sous-sortes est la modélisation la plus proche du modèle mathématique usuel : en effet, on considère habituellement qu'une fonction telle que la division arithmétique est définie sur $\mathbb{N}\times(\mathbb{N}\setminus\{0\})$, et non sur $\mathbb{N}\times\mathbb{N}$; "$0 \; div \; 0$" est une expression qui n'a mathématiquement *aucun sens*.

Ainsi que Michel Bidoit l'évoque dans sa thèse [22], les problèmes rencontrés pour définir les sous-sortes et les termes d'erreur sont proches. On peut considérer que la définition d'une sous-sorte comme domaine de définition d'un opérateur *cache* à l'utilisateur les valeurs erronées que pourrait prendre cet opérateur. La parenté entre les deux approches est encore plus évidente si l'on n'autorise les déclenchements d'exception que par le biais de règles de précondition (comme en LPG 1.8) :

[63] Étant donné une signature $\Sigma = (S, \Omega)$, et "$\succsim_\Omega$" un préordre sur les opérateurs, appelé **relation de précédence**, l'**ordre lexicographique sur les chemins** (*lexicographic path ordering*), ou ordre de Kamin et Lévy [119], est défini comme suit, pour $t, u \in T_\Sigma(V)$, avec $t \equiv f(t_1, \ldots, t_m)$ et $u \equiv g(u_1, \ldots, u_n)$:

$$t \succsim_{lpo} u \overset{\text{déf}}{\iff} \begin{cases} (\exists i, 1 \leqslant i \leqslant m), t_i \succsim_{lpo} u \\ \vee \\ (f \succsim_\Omega g \wedge (\forall j, 1 \leqslant j \leqslant n), t \succ_{lpo} u_j) \\ \vee \\ (f \approx_\Omega g \wedge (t_1, \ldots, t_m) \succ_{lpo}^{*} (u_1, \ldots, u_n) \wedge (\forall j, 2 \leqslant j \leqslant n), t \succsim_{lpo} u_j) \end{cases}$$

où "$\succsim_{lpo}^{*}$" est l'extension lexicographique à gauche de "$\succsim_{lpo}$", définie comme suit :

$$(t_1, \ldots, t_m) \succ_{lpo}^{*} (u_1, \ldots, u_n) \overset{\text{déf}}{\iff} (\exists i, 1 \leqslant i \leqslant n), (\forall j < i), t_i \approx_{lpo} u_i \wedge t_j \succ_{lpo} u_j$$

et où "$\approx_\Omega$" et "$\approx_{lpo}$" désignent bien sûr les fermetures symétriques de "$\succsim_\Omega$" et de "$\succsim_{lpo}$".

Cet ordre est utilisé pour prouver la terminaison de systèmes de réécriture.

cela revient à définir un opérateur sur la sous-sorte formée des éléments pour lesquels toutes les préconditions échouent.

Une définition possible pour une sous-sorte est de poser une restriction sur les constructeurs. On peut rattacher à ce procédé les sous-sortes engendrées par une famille de constructeurs : on interdit *ipso facto* l'usage des autres constructeurs. Ainsi que nous l'avons vu au §0.1.2.5 à propos des contraintes de sortes, on peut aussi définir des sous-sortes au moyen de prédicats, comme en EQLOG [83].

Nous remarquerons cependant que cette approche est davantage orientée vers la spécification que vers la programmation, car cela revient à éliminer les cas d'erreur avant de commencer le calcul proprement dit, alors qu'en programmation, les deux objectifs sont parfois menés simultanément : voir à ce sujet la remarque du §4.2.2.

4.4.3 Opérateurs "OK" et opérateurs "erreur"

Joseph Goguen introduit dans [71] les **présentations avec erreurs**, de la forme (S, Ω, Ξ, K, E), en partageant les opérateurs en opérateurs "OK" (Ω) et opérateurs "erreur" (Ξ). Par suite, un terme est "OK" s'il n'est constitué que d'opérateurs "OK", c'est un terme d'erreur s'il est formé d'un opérateur "erreur" ou d'un opérateur "OK" dont un des arguments est un terme d'erreur. Les parties gauche et droite d'une équation "OK" (appartenant à K) sont des termes "OK" et une telle équation ne peut s'appliquer qu'à un terme "OK". Les autres équations sont dites équations "erreur" et forment l'ensemble E. La propagation des erreurs est implicite.

Cette technique a été mise en œuvre dans la première version d'OBJ [82]. La facilité de propagation implicite rend les spécifications lisibles. Par contre, il est impossible de récupérer des termes d'erreur.

Au niveau sémantique, Joseph Goguen introduit les **algèbres d'erreur**. Une algèbre d'erreur A sur une signature d'erreur (S, Ω, Ξ) est une (S, Ω, Ξ)-algèbre dans laquelle le support de toute sorte est l'union disjointe d'éléments "OK" et d'éléments "erreur" :

$$A_s = K_s \uplus E_s$$

et telle que :

(i) $(\forall \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}), (\forall u_i \in A_{s_i}, 1 \leqslant i \leqslant n), ((\exists j, 1 \leqslant j \leqslant n), u_j \in E_{s_j} \Longrightarrow A_\omega(u_1, \ldots, u_n) \in E_s)$

(ii) $(\forall \xi \in \Xi_{s_1 \times \cdots \times s_n \to s}), (\forall u_i \in A_{s_i}, 1 \leqslant i \leqslant n), A_\xi(u_1, \ldots, u_n) \in E_s$

Une algèbre d'erreur A satisfait une présentation avec erreurs si pour toute assignation ν à valeurs dans A, les propriétés suivantes sont vérifiées :

$$\begin{array}{lll}(\forall (t_1 == t_2) \in K), & (\nu(t_1) \in K_{sort(t_1)} \wedge \nu(t_2) \in K_{sort(t_2)}) & \wedge \quad \nu(t_1) = \nu(t_2) \\ (\forall (t_1 == t_2) \in E), & (\nu(t_1) \in E_{sort(t_1)} \vee \nu(t_2) \in E_{sort(t_2)}) & \wedge \quad \nu(t_1) = \nu(t_2)\end{array}$$

Suivent dans [71] l'introduction des morphismes entre algèbres d'erreur et de la catégorie des algèbres d'erreur qui satisfont une présentation. Mais cette catégorie est trop large pour admettre toujours un objet initial. Pour garantir l'existence de cet objet initial, Joseph Goguen considère les présentations avec erreurs pour lesquelles il existe au moins un opérateur d'erreur dans chaque équation de E. Alors, la catégorie des algèbres d'erreur qui satisfont cette présentation et qui sont telles qu'il est possible, avant d'effectuer une évaluation, de décider si le résultat sera un terme "OK" ou un terme d'erreur, admet une algèbre initiale. Il ne fait cependant aucun doute que cette dernière condition est très restrictive.

4.4.4 Opérateurs *safe* et opérateurs *unsafe*

La solution proposée dans [70] consiste à déclarer *unsafe* les opérateurs qui peuvent retourner des erreurs, les autres opérateurs étant *safe* par défaut. Il résulte de ce choix qu'un terme dont tous les opérateurs sont *safe* est "OK". Les variables sont partagées en deux groupes : les variables v, telles que $ok(v) = true$, pouvant être substituées par des termes "OK", et les variables $v+$, telles que $ok(v+) = false$, pouvant être substituées par des termes d'erreur. Il n'y a ni distinction entre équations "OK" et équations erreur, ni propagation implicite des erreurs. Par contre, une récupération d'erreur se spécifie au moyen d'une équation dont l'une des parties est un terme d'erreur et l'autre partie un terme "OK".

L'avantage de ce formalisme : l'existence d'une algèbre initiale est garantie. Son grand inconvénient : il s'adapte très mal aux cas où un constructeur est partiel. Si nous sommes obligés de déclarer *unsafe* le constructeur "//" des rationnels (cf. exemple 4.1), quels seront les termes "OK" de sorte *Rat* ?

4.4.5 Exception-spécifications et exception-algèbres

Constatant les insuffisances des précédents formalismes, Gilles Bernot a développé dans sa thèse [10] un nouveau formalisme fondé sur une distinction entre exception et erreur. Il poursuit un double but de fournir une spécification très fine des états "anormaux" d'une évaluation — spécification utilisable pour la description d'implantations abstraites en présence d'exceptions —, en même temps qu'une sémantique solide et la possibilité d'étudier des propriétés telles que la complétude suffisante et la consistance hiérarchique à l'intérieur de ce nouveau formalisme. Ainsi, il n'impose pas une partition stricte des termes en termes "OK" et termes exceptionnels. Un terme peut être exceptionnel s'il est *étiqueté* par une exception, mais sa *valeur* peut être "OK", auquel cas c'est qu'il a été récupéré. De même, un terme qui n'est pas étiqueté par une exception n'est pas nécessairement "OK", il peut s'agir d'une application erronée d'un opérateur qui n'est pas complètement défini.

Au niveau syntaxique, Gilles Bernot introduit les **exception-spécifications**, définies par :

$$\langle S, \Sigma, L, Ok\text{-}Frm, Ok\text{-}Ax, Lbl\text{-}Ax, Gen\text{-}Ax \rangle$$

où :

- S est un ensemble de sortes ;
- Σ un ensemble d'opérateurs indicés par des éléments de $S^* \times S$;
- L un ensemble d'*étiquettes* d'exception ;
- *Ok-Frm* (*OK forms*) : axiomes spécifiant les valeurs que l'on considère "OK" ;
- *Ok-Ax* (*OK axioms*) : équations (conditionnelles) entre termes "OK" ;
- *Lbl-Ax* (*labelling axioms*) : ces axiomes regroupent les cas où l'on souhaite attacher une étiquette d'exception au résultat d'un opérateur appliqué à des termes "OK" ;
- *Gen-Ax* (*generalized axioms*) : traitement des termes exceptionnels — à noter que les exceptions se propagent implicitement, sauf si elles sont récupérées.

Au niveau sémantique, les modèles associés à une exception-spécification sont les **exception-algèbres.** Une exception-algèbre sur une exception-signature $\langle S, \Sigma, L\rangle$ est une algèbre A munie pour chaque étiquette l de $L \cup \{OK\}$ d'un ensemble A_l représentant les valeurs étiquetées par l. Ainsi, les éléments de A_{OK} sont les valeurs "OK" de l'algèbre A. Ces ensembles A_l peuvent être hétérogènes, c'est-à-dire recouper les supports de plusieurs sortes.

Une exception-algèbre A **valide** *Ok-Frm* si toute assignation d'un terme "OK" est un élément de A_{OK}. Elle valide *Lbl-Ax* si, pour toute étiquette l de L, toute assignation d'un terme étiqueté par l appartient à A_l. Les validations de *Ok-Ax* et *Gen-Ax* sont analogues aux validations d'équations conditionnelles dans le cas "classique".

Sont ensuite définis les **exception-morphismes**, morphismes qui préservent les étiquettes. Les exception-algèbres qui satisfont une exception-spécification, munis des exception-morphismes, forment les objets et les flèches d'une catégorie admettant un objet initial.

Ce formalisme est le plus complet que nous connaissons à ce jour, mais sa lourdeur le rend difficile à intégrer, surtout dans un langage que l'on chercherait à orienter vers la programmation.

4.4.6 Algèbres avec sortes ordonnées

Les sortes ordonnées (cf. §0.1.2) apportent une solution au problème des opérateurs partiels. En effet, outre les possibilités de coercition entre des sortes telles que *Nat* et *Rat*, signalées au §0.1.2, un des principaux apanages des sortes ordonnées est de permettre, à travers un même moule théorique qui est l'ordre partiel sur les sortes, aussi bien la définition de sous-sortes que celle de sur-sortes, qui dans le cas présent regroupent en sus les opérateurs représentant les erreurs. Les deux spécifications suivantes des séquences génériques le montrent.

Exemple 4.5 *Deux spécifications de l'opérateur tail sur les séquences.*

```
obj SEQ[X :: TRIV] is
   sorts Seq NeSeq .
   subsorts NeSeq < Seq .
   op nil   :            -> Seq .
   op __    : Elt    Seq -> NeSeq .
   op tail_ : NeSeq      -> Seq .
   var e    : Elt .
   var sq   : Seq .
   eq tail (e sq) = sq .
jbo
```

```
obj SEQERR[X :: TRIV] is
   sorts Seq SeqErr .
   subsorts Seq < SeqErr .
   op nil   :                   -> Seq .
   op __    : Elt    Seq        -> Seq .
   op __    : Elt    SeqErr     -> SeqErr .
   op err   :                   -> SeqErr .
   op tail_ : SeqErr            -> SeqErr .
   var e    : Elt .
   var sq   : Seq .
   eq       e err = err .
   eq    tail nil = err .
   eq tail (e sq) = sq .
   eq    tail err = err .
jbo
```

Dans la spécification de droite, la valeur d'erreur est intégrée à la sorte `SeqErr` et permet d'écrire une définition complète de l'opérateur `tail`. Dans celle de gauche, l'utilisation de rétractions (cf. §0.1.2.4) équivaut à des tests de sorte à l'exécution et permet alors de n'appliquer l'opérateur `tail` qu'à des séquences non vides. Ainsi :

```
tail tail (1 2 nil)
```

est *parsé* comme suit :

$$\texttt{tail}\ r_{Seq \to NeSeq}\texttt{(tail (1 2 nil))}$$

et est réduit à `nil` selon le procédé vu au §0.1.2.4.

Outre l'avantage de posséder une sémantique d'algèbre initiale, cette approche permet d'éliminer le problème de la récupération des erreurs. Si l'on adopte un principe de spécification avec sur-sortes incluant les opérateurs simulant les erreurs, il suffit de définir les opérateurs de façon complète sur ces sur-sortes. C'est la méthode que suggère Joseph Goguen dans [75].

Cette approche n'est cependant pas exempte d'inconvénients. Le premier : les spécifications obtenues ne peuvent se défendre d'une certaine complexité. Les deuxième et troisième inconvénients sont liés à l'approche suivie : par sous-sortes, ou par sur-sortes incluant les termes d'erreur (`NeSeq` et `Seq`, `Seq` et `SeqErr` dans l'exemple précédent). Dans le premier cas, à moins que l'utilisateur prenne lui-même en charge la spécification des opérateurs de rétraction, il n'existe aucun moyen de récupérer des évaluations de termes dans lesquels un opérateur est appliqué à un sous-terme dont la plus petite sorte est "trop grande". Si l'approche par sur-sortes rend l'écriture de récupérateurs beaucoup plus aisée, en revanche, l'utilisateur doit écrire toutes les équations de propagation des valeurs d'erreur — dans notre exemple, l'équation (`e err = err`).

En fait, l'approche par sortes ordonnées permet de spécifier plus finement les opérateurs, et est certainement très adaptée aux preuves — d'ailleurs, OBJ3 est avant tout un interpréteur de la logique avec sortes ordonnées et décomposition modulaire, mais il se révèle aussi un bon outil d'aide à la preuve de théorèmes [74] —, tandis que les exceptions, telles que nous les avons définies, sont préférables d'un point de vue pratique, afin de modéliser la notion d'état anormal. Ce que nous allons voir maintenant, c'est comment cette seconde approche rejoint la première.

4.5 Sémantique algébrique

La sémantique que nous proposons est fondée sur le principe d'associer à toute sorte s le support décrit par l'interprétation des termes de sorte s, puis un sur-ensemble comprenant, en sus des éléments du précédent, l'interprétation de tous les termes exceptionnels pouvant être utilisés en lieu et place de termes de sorte s. À noter que dans cette sémantique, un terme d'exception n'a donc pas toujours la même interprétation : elle dépend de la sorte du terme qu'il remplace.

Une telle démarche évoque les sortes ordonnées. Notre idée de base revient en effet à considérer pour chaque sorte s introduite par l'utilisateur, deux sortes s et $\widehat{s}$, s qualifiant les termes non exceptionnels, $\widehat{s}$ englobant les termes exceptionnels et non exceptionnels. Nous allons préciser ce que sont les présentations avec exceptions, et les définir comme des cas particuliers de présentations avec sortes ordonnées. Nous verrons par la suite que le comportement décrit précédemment de façon opérationnelle est bel et bien simulé par la présentation avec sortes ordonnées obtenue.

4.5.1 Présentation avec exceptions : cas particulier de présentation avec sortes ordonnées

Définition 4.6 *Une* **présentation avec exceptions** *est un sextuplet de la forme* $(S, \Gamma, \Omega, \Phi, Pr, E)$ *où* S *est un ensemble non vide de sortes,* Γ *un ensemble d'opérateurs constructeurs indicés par* $S^* \times S$*,* Ω *un ensemble d'opérateurs non constructeurs indicés par* $S^* \times S$*,* Φ *un ensemble d'opérateurs d'exception indicés par* S^**,* Pr *un ensemble de règles de précondition sur* Γ*, et* E *un ensemble d'équations conditionnelles.*

Par convention, Pr contient uniquement les préconditions des opérateurs de Γ, c'est-à-dire des constructeurs. Nous considérons que les préconditions des opérateurs de Ω sont rangées dans l'ensemble E qui est, rappelons-le, un ensemble d'équations conditionnelles.

Sans perte de généralité, nous posons qu'une règle de précondition de Pr se présente sous la forme suivante :

$$c(x_1, \ldots, x_n) \mid g(\ldots) \longrightarrow \phi(\ldots)$$

où $c \in \Gamma_{s_1 \times \cdots \times s_n \to s}$, et les $(x_i)_{1 \leqslant i \leqslant n}$ sont des variables de sortes respectives s_i, pour $1 \leqslant i \leqslant n$.

Remarque La convention des (x_i) variables n'occasionne aucune perte de généralité, car il est toujours possible pour chaque constructeur, de définir un discriminant de codomaine *Bool*, et des sélecteurs. Ainsi, la règle de précondition du constructeur des rationnels positifs (cf. exemple 4.1) peut s'écrire :

$$i \mathbin{//} j \mid zero?(j) \longrightarrow !zero\text{-}divide_{PRat}$$

Ceci revient à adapter au langage de spécification la démarche de compilation des sortes en FP2 (cf. §2.1.2.2).

Deux hypothèses supplémentaires :

- le support du type *Bool* est isomorphe à $\{true, false\}$, avec $true \not\equiv false$,
- toute évaluation d'un terme booléen compris comme une condition (prémisse d'une équation conditionnelle, précondition d'une règle de précondition, condition d'un schéma *if-then-else*) ne retourne *jamais* de terme exceptionnel.

On aura reconnu qu'il s'agit là de conventions proches de la complétude et la consistance par rapport aux booléens, vues à propos de la réécriture conditionnelle (cf. définition 0.39).

Une telle présentation définit une présentation avec sortes ordonnées $(\overline{S}, \leq, \overline{\Omega}, \overline{E}, \overline{C})$ où :

$$\begin{array}{rcl}
\overline{S} & = & S \cup \{\widehat{s} \mid s \in S\} \\
\leq & = & \{(s, \widehat{s}) \mid s \in S\}^{=} \\
\overline{\Omega} & = & \{\gamma : \widehat{s_1} \times \cdots \times \widehat{s_n} \to \widehat{s} \mid \gamma \in \Gamma_{s_1 \times \cdots \times s_n \to s}, s_1, \ldots, s_n, s \in S\} \cup \\
& & \{\omega : \widehat{s_1} \times \cdots \times \widehat{s_n} \to \widehat{s} \mid \omega \in \Omega_{s_1 \times \cdots \times s_n \to s}, s_1, \ldots, s_n, s \in S\} \cup \\
& & (\bigcup_{s \in S} \{\phi_{\widehat{s}} : \widehat{s_1} \times \cdots \times \widehat{s_n} \to \widehat{s} \mid \phi \in \Phi_{s_1 \times \cdots \times s_n}, s_1, \ldots, s_n \in S\}) \\
\overline{E} & = & E' \cup Pr' \cup rec(E) \cup rec(Pr) \cup abs(\overline{\Omega}) \\
\overline{C} & = & \{sort\text{-}constraint(c) \mid c \in \Gamma\}
\end{array}$$

où "$\mathcal{G}^{=}$" désigne la fermeture réflexive de la relation représentée par le graphe $\mathcal{G}$.

Autrement dit, pour tout élément s de S, nous créons une sur-sorte $\widehat{s}$ ($s \leq \widehat{s}$), et tous les opérateurs de Γ et de Ω sont définis par rapport à ces sur-sortes. À chaque opérateur d'exception, nous lui associons un ensemble d'opérateurs dont le domaine est construit à l'aide des sur-sortes, et dont les codomaines parcourent toutes les sur-sortes introduites. Nous venons ainsi de traduire que tout opérateur d'exception, avec un indice *ad hoc*, peut être utilisé pour construire des éléments de sorte $\widehat{s}$, avec $s \in S$.

Les déclarations de variables sont invariantes. Toute équation de E' est déduite d'une équation de E par indiçage des opérateurs d'exception par la sur-sorte attendue, et par remplacement de chaque

occurrence de récupérateur par un terme préfixé par un opérateur caché. Ainsi, soit un terme t_0 de sorte s, et soit le récupérateur (supposé défini pour tous les opérateurs d'exception) :

$$\begin{array}{lllll} \textbf{rec} & t_0 & & & \\ & \textbf{when} & \phi_{\hat{s}}^1(\tau_{1,1},\ldots,\tau_{1,\varpi_1}) & \textbf{then} & t_1 \\ & \vdots & & & \\ & \textbf{when} & \phi_{\hat{s}}^n(\tau_{n,1},\ldots,\tau_{n,\varpi_n}) & \textbf{then} & t_n \\ \textbf{endrec} & & & & \end{array}$$

avec $\varpi_i = \text{arity}(\phi_{\hat{s}}^i), (1 \leqslant i \leqslant n)$.

Remarque Il est toujours possible de définir complètement un récupérateur sous cette forme. Si la clause "**otherwise**" est présente, il suffit de la développer d'après les opérateurs d'exception qui ne sont pas apparus précédemment. Si cette clause est absente, on introduit, pour chaque opérateur d'exception $\phi_{\hat{s}}^j$ absent dans le récupérateur :

$$\textbf{when } \phi_{\hat{s}}^j(\ldots) \textbf{ then } \phi_{\hat{s}}^j(\ldots)$$

c'est-à-dire que l'on traduit que le récupérateur équivaut à l'identité dans ce cas.

L'ensemble des variables libres[64] de ce récupérateur est :

$$var(t_0) \cup (\bigcup_{1 \leqslant i \leqslant n} (var(t_i) \setminus var(\phi_{\hat{s}}^i(\tau_{1,1},\ldots,\tau_{1,\varpi_1}))))$$

Ordonnons-le en une famille $(x_1,\ldots,x_q)$, les sortes respectives étant $s_1,\ldots,s_q$. Ce terme est alors remplacé par le terme $\eta(t_0,x_1,\ldots,x_q)$, avec :

$$\begin{array}{lllllllllll} \eta & : & s & \times & s_1 & \times & \cdots & \times & s_q & \rightarrow & s \\ \eta & : & \hat{s} & \times & s_1 & \times & \cdots & \times & s_q & \rightarrow & \hat{s} \end{array}$$

opérateurs cachés définis par (x étant une variable de sorte s) :

$$\begin{array}{rcl} \eta(x,x_1,\ldots,x_q) & == & x \\ \eta(\phi_{\hat{s}}^1(\tau_{1,1},\ldots,\tau_{1,\varpi_1}),x_1,\ldots,x_q) & == & t_1 \\ & \vdots & \\ \eta(\phi_{\hat{s}}^n(\tau_{n,1},\ldots,\tau_{1,\varpi_n}),x_1,\ldots,x_q) & == & t_n \end{array} \quad (4.4)$$

Remarquons que :

- l'équation (4.4) traduit que les termes non exceptionnels sont invariants par la récupération η,
- η est totalement défini (c'est-à-dire qu'il n'existe aucun terme irréductible préfixé par η), il est par conséquent possible de le cacher sans perturber la propriété d'existence éventuelle d'une algèbre initiale satisfaisant la présentation.

Ainsi, toute équation ($t_1 == t_2$ if c) de E devient par ce procédé une équation ($t_1' == t_2'$ if c'). L'hypothèse posée sur les prémisses des équations conditionnelles assure que le résultat de l'évaluation de c est bien de sorte *Bool*, et non un terme exceptionnel de sorte $\widehat{Bool}$.

[64]Dans ce cas de figure, il s'agit des variables qui ne sont pas introduites par la partie gauche du récupérateur ou par des formes *let* présentes à l'intérieur des termes $t_1,\ldots,t_n$.

Les équations caractérisant les opérateurs cachés qui correspondent aux récupérateurs présents dans les équations de E forment l'ensemble $rec(E)$. Les ensembles d'équations Pr' et $rec(Pr)$ sont construits de manière analogue.

$abs(\overline{\Omega})$ représente la traduction en équations de la règle d'absorption. Pour tout $f \in \overline{\Omega}_{\widehat{s_1} \times \cdots \times \widehat{s_n} \to \widehat{s}}$, on construit pour les places 1 à n les équations suivantes :

$$f(x_1, \ldots, x_{i-1}, \phi_{\widehat{s_i}}(\ldots), z_{i+1}, \ldots, z_n) == \phi_{\widehat{s}}(\ldots) \text{ avec : } \begin{cases} sort(x_j) = s_j & (1 \leqslant j < i) \\ sort(z_j) = \widehat{s_j} & (i < j \leqslant n) \end{cases}$$

pour chaque $\phi \in \Phi$.

L'impossibilité de substituer une variable x_j de sorte s_j par un terme exceptionnel assure que l'absorption a bien lieu en faveur du terme exceptionnel situé le plus à gauche, en l'occurrence au rang i. Remarquons que les opérateurs issus de Φ admettent eux aussi des équations analogues, et qu'aucune de ces équations introduites par $abs(\overline{\Omega})$ n'est superposable avec une équation de Pr ou une équation de E. Par conséquent, l'ajout des équations de $abs(\overline{\Omega})$ ne modifie pas la relation de congruence entre termes non exceptionnels.

Nous abordons à présent le dernier point syntaxique : comment engendrer, pour une sorte s, des termes non exceptionnels. $\overline{E}$ contenant les règles de précondition de Γ, ce qui reste à formuler, c'est que les termes correspondant à l'échec de toutes les préconditions ne sont pas des termes exceptionnels : ceci s'effectue à l'aide de contraintes de sortes.

Soient $c \in \Gamma$, avec Pr_c les règles de précondition qui s'appliquent à l'opérateur c. Pour c : $\widehat{s_1} \times \cdots \times \widehat{s_n} \to \widehat{s}$, si Pr_c est constitué par :

$$\begin{array}{lcl} c(x_1, \ldots, x_n) \mid g_1(\ldots) & \longrightarrow & \phi_1(\ldots) \\ \vdots & & \vdots \\ c(x_1, \ldots, x_n) \mid g_p(\ldots) & \longrightarrow & \phi_p(\ldots) \end{array}$$

avec $sort(x_i) = s_i$, pour $1 \leqslant i \leqslant n$, alors $sort\text{-}constraint(c)$ est la règle :

$$\text{as } c(x_1, \ldots, x_n) \text{ if } \bigwedge_{1 \leqslant i \leqslant p} (\neg g_i(\ldots))$$

si $Pr_c = \varnothing$, alors $sort\text{-}constraint(c)$ est la règle :

$$\text{as } s : c(x_1, \ldots, x_n) \text{ if } true$$

Rappelons que par hypothèse, les règles de préconditions sont telles que pour toute précondition $g_i(\ldots)$, avec :

$$g_i : s_{i,1} \times \cdots \times s_{i,\varpi_i} \to Bool \qquad (1 \leqslant i \leqslant p)$$

nous avons :

$$(\forall y_{i,j}, sort(y_{i,j}) = s_{i,j}, 1 \leqslant j \leqslant p), g_i(y_{i,1}, \ldots, y_{i,\varpi_i}) \equiv true \vee g_i(y_{i,1}, \ldots, y_{i,\varpi_i}) \equiv false$$

Ceci permet l'utilisation de l'opérateur de négation, et la contrainte de sorte résultante vérifie elle aussi la même propriété :

$$(\bigwedge_{1 \leqslant i \leqslant p} (\neg g_i(y_{i,1}, \ldots, y_{i,\varpi_i})) \equiv true) \vee (\bigwedge_{1 \leqslant j \leqslant p} (\neg g_i(y_{i,1}, \ldots, y_{i,\varpi_i})) \equiv false)$$

4.5.2 Propriétés de la présentation obtenue — Niveau sémantique

Notons $\Sigma = (\overline{S}, \leq, \overline{\Omega})$. Σ est régulière : les deux profils d'un récupérateur vérifient la condition. Quant aux autres opérateurs, ils n'admettent qu'une seule déclaration de profil[65].

Les modèles d'une telle présentation sont donc des Σ-algèbres $A = (A_{\overline{S}}, A_{\overline{\Omega}})$ tels que pour $s \in S$:

- A_s contient les interprétations des termes de sorte s,
- $A_{\hat{s}}$ contient les interprétations des termes de sorte s et des termes exceptionnels pouvant être employés en lieu et place des termes de sorte s.

La régularité de la signature Σ obtenue assure que la catégorie des Σ-algèbres admet un objet initial.

La présentation obtenue est cohérente, par construction de "$\leq$" — $(\overline{S}, \leq)$ admet autant de composantes connexes qu'il existe d'éléments dans S. Par suite, la catégorie des Σ-algèbres qui satisfont $\overline{E}$ et $\overline{C}$ admet un objet initial.

Exemple 4.7 *Soit la présentation suivante :*

presentation $\mathcal{P}$ **is**
  sorts $= \{Bool, Nat\}$
  opns

true, false	:	–		→	*Bool*
not	:	*Bool*		→	*Bool*
and, or	:	*Bool*	× *Bool*	→	*Bool*
0	:	–		→	*Nat*
succ	:	*Nat*		→	*Nat*
div	:	*Nat*	× *Nat*	→	*Nat*
zero?	:	*Nat*		→	*Bool*

  vars ...
  prcds
    <> *i div j* | *zero?(j)* ==> *!zero-divide*
  eqns
    ⟨⟨équations définissant *not, and, or, div*⟩⟩
    <> *zero?(0)* ==> *true*
    <> *zero?(succ(i))* ==> *false*
end $\mathcal{P}$

La présentation avec sortes ordonnées correspondante est telle que son algèbre des termes est :

$$
\begin{aligned}
\mathcal{T}_{Bool} &= \{true, false\} \\
\mathcal{T}_{Nat} &= \{0, succ(0), succ(succ(0)), \ldots\} \\
\mathcal{T}_{\widehat{Bool}} &= \mathcal{T}_{Bool} \cup \{!zero\text{-}divide_{Bool}\} \\
\mathcal{T}_{\widehat{Nat}} &= \mathcal{T}_{Nat} \cup \{!zero\text{-}divide_{Nat}\}
\end{aligned}
$$

[65] Pour des raisons de simplicité, la surcharge de FP2 n'est pas considérée ici.

Exemple 4.8 (Suite de l'exemple 4.7) *Spécifions une récupération de l'exception "!zero-divide" et explicitons son traitement.*

presentation $\mathcal{P}'$ **is** $\mathcal{P} \uplus$
  opns *rec-div* : *Nat* × *Nat* → *Nat*
  vars ...
  eqns
    <> *i rec-div j* == **rec** *i div j*
        when !*zero-divide* **then** 0
      endrec
end $\mathcal{P}'$

La présentation avec sortes ordonnées est telle que les déclarations des profils des opérateurs sont :

$$\begin{array}{rclcccccc} rec\text{-}div & : & & & \widehat{Nat} & \times & \widehat{Nat} & \rightarrow & \widehat{Nat} \\ \eta & : & Nat & \times & Nat & \times & Nat & \rightarrow & Nat \\ \eta & : & \widehat{Nat} & \times & Nat & \times & Nat & \rightarrow & \widehat{Nat} \end{array}$$

où η est un opérateur caché. Voici à présent les équations :

$$\begin{array}{rcl} i\ rec\text{-}div\ j & == & \eta(i\ div\ j) \\ i\ rec\text{-}div\ !zero\text{-}divide_{Nat} & == & !zero\text{-}divide_{Nat} \\ !zero\text{-}divide_{Nat}\ rec\text{-}div\ z & == & !zero\text{-}divide_{Nat} \\ \eta(k,i,j) & == & k \\ \eta(!zero\text{-}divide_{Nat},i,j) & == & 0 \end{array}$$

i, j, k étant des variables de sorte *Nat*, et z une variable de sorte $\widehat{Nat}$.

Si un constructeur c, de profil $\widehat{s_1} \times \cdots \times \widehat{s_n} \rightarrow \widehat{s}$ admet pour seule contrainte de sorte :

$$\text{as } s : c(x_1, \ldots, x_n) \text{ if } true$$

avec $sort(x_i) = s_i$, pour $1 \leqslant i \leqslant n$, ceci revient à définir deux profils pour c :

$$\begin{array}{rcccccc} c: & s_1 & \times & \cdots & \times & s_n & \rightarrow & s \\ c: & \widehat{s_1} & \times & \cdots & \times & \widehat{s_n} & \rightarrow & \widehat{s} \end{array}$$

Le second profil de c est utilisé pour traduire la propagation des opérateurs d'exception dans $abs(\overline{\Omega})$. Dans ce cas de contrainte de sorte triviale, la présence du premier profil permet d'affirmer que si tous les constructeurs d'une sorte s sont totaux, alors toute interprétation de s dans une Σ-algèbre coïncide avec la définition sans exceptions. Il en résulte que si $\Phi = \emptyset$, alors on retrouve le formalisme et la théorie des algèbres hétérogènes classiques, toute Σ-algèbre A étant dans ce cas telle que pour tout $s \in S, A_s = A_{\widehat{s}}$.

Le système de réécriture obtenu en orientant les équations de $\overline{E}$ décroît les sortes. Cette propriété est aisément vérifiable pour les règles de réécriture définissant un récupérateur et les règles simulant l'absorption des opérateurs d'exception. Pour les règles qui proviennent d'équations de E, elles sont de la forme :

$$f(t_1, \ldots, t_n) \longrightarrow t$$

avec $LS(t_i) = s_i$, pour $1 \leqslant i \leqslant n$. Dans $\overline{\Omega}$, le profil de f est de la forme $\widehat{s_1} \times \cdots \times \widehat{s_n} \to \widehat{s}$, avec $s_1, \ldots, s_n, s \in S$. Par conséquent :

$$LS(\overline{f(x_1, \ldots, x_n)}) = \widehat{s}$$

Que $LS(t)$ soit égal à s ou $\widehat{s}$, la propriété de décroissance des sortes est vérifiée.

Nous pouvons par conséquent énoncer le théorème suivant :

Théorème 4.9 *Toute présentation avec exceptions se développe en une présentation avec sortes ordonnées qui admet une algèbre initiale. De plus, le système de réécriture obtenu dans la présentation avec sortes ordonnées vérifie la propriété de Church-Rosser.*

4.5.3 Niveau opérationnel

Il est assez facile de voir que la règle d'absorption est une réalisation des équations de l'ensemble $abs(\overline{\Omega})$. De même, il est possible d'implanter un opérateur caché issu d'un récupérateur en comprenant l'opérateur caché comme un système de réécriture dont on n'applique qu'un pas au plus (cf. §4.2.3). Par conséquent :

Théorème 4.10 *La règle d'absorption et la récupération de termes exceptionnels sont des implantations correctes des présentations avec exceptions.*

Par "implantation correcte", nous entendons le fait que la sémantique opérationnelle que nous avons décrite au §4.2 donne les mêmes résultats que la réécriture des termes de la présentation avec sortes ordonnées qui correspond à une présentation avec exceptions.

4.5.4 Remarques et compléments

4.5.4.1 Introduction de nouvelles définitions

Lors de l'introduction d'une nouvelle sorte, il est bien sûr nécessaire d'indicer les opérateurs de Φ avec la nouvelle sorte. De même, l'introduction d'un nouvel opérateur d'exception doit s'accompagner de la spécification de tous les opérateurs cachés de récupération pour ce nouvel opérateur, ainsi que de la spécification de toutes les règles d'absorption propageant les termes formés avec le nouvel opérateur d'exception, ceci pour tous les opérateurs précédemment définis.

4.5.4.2 Exceptions et généricité

Notre définition des présentations avec exceptions s'étend au cas générique. Il est certain que l'on obtient alors des présentations avec sortes ordonnées très complexes. Sans l'étudier complètement, nous allons néanmoins donner ci-après les principes de cette extension afin de montrer qu'elle ne pose pas de problème au niveau théorique.

L'idée d'associer à une sorte s spécifiée par l'utilisateur, deux sortes s et $\widehat{s}$, telles que $s \leq \widehat{s}$, nous la reconduisons pour les sortes génériques, et également pour les sortes introduites au moyen de présentations paramètres (propriétés), ce qui modifie également la signature de ces dernières. De manière analogue, nous indiçons les opérateurs d'exception, éventuellement par des sortes formelles, développons sous forme d'équations la propagation des termes exceptionnels, et traitons les récupérateurs au moyen d'opérateurs auxiliaires cachés.

En ce qui concerne l'instanciation des unités génériques, une conséquence de notre méthode est qu'à toute liaison $[t \mapsto s]$ spécifiée par l'utilisateur, doit correspondre une double liaison :

$$[t \mapsto s, \widehat{t} \mapsto \widehat{s}]$$

qui assure par construction la compatibilité du morphisme de présentation avec l'ordre partiel sur les sortes. Il est également indispensable d'ajouter une liaison supplémentaire pour chaque opérateur d'exception indicé par une sorte formelle ou par une sorte paramétrée par au moins une sorte formelle à l'intérieur de la présentation paramétrée. Ainsi, les liaisons :

$$[t \mapsto Nat, \widehat{t} \mapsto \widehat{Nat}, !zero\text{-}divide_{Seq[t]} \mapsto !zero\text{-}divide_{Seq[Nat]}]$$

permettent d'utiliser les règles suivantes, où t est une sorte formelle :

$$\begin{array}{rcll} !zero\text{-}divide_t <+ z & \longrightarrow & !zero\text{-}divide_{Seq[t]} & (z : \widehat{Seq}[t]) \\ reverse(!zero\text{-}divide_{Seq[t]}) & \longrightarrow & !zero\text{-}divide_{Seq[t]} & \end{array}$$

pour toute instanciation des séquences génériques. Ainsi :

$$\begin{array}{rcl} reverse([1\ div\ 0]) & \longrightarrow & reverse([!zero\text{-}divide_{Nat}]) \\ & \longrightarrow & reverse(!zero\text{-}divide_{Seq[Nat]}) \\ & \longrightarrow & !zero\text{-}divide_{Seq[Nat]} \end{array}$$

Du point de vue de la propagation et de la récupération des exceptions génériques, signalons que notre définition des opérateurs d'exceptions et des récupérateurs garantit contre les cas signalés au §4.2.4 (ils sont tous deux refusés). Remarquons également que l'on retrouve la règle appliquée empiriquement en LPG, à savoir que seule une instanciation qui correspond à un renommage des paramètres formels d'un opérateur d'exception générique peut figurer en partie gauche d'une clause de récupération : c'est une conséquence de la définition d'un récupérateur au moyen d'un opérateur auxiliaire caché.

Par conséquent, l'extension aux présentations avec exceptions d'une propriété étudiée dans le cadre des présentations génériques n'est pas rendue impossible au départ, du fait de la définition des présentations avec exceptions. Les sémantiques proposées permettent, pour l'étude de telle ou telle propriété, de choisir un cadre de travail parmi des niveaux emboîtés. Le niveau le plus restreint est celui des présentations multi-sortes. Viennent ensuite, dans l'ordre, les présentations génériques, puis les présentations avec sortes ordonnées et généricité.

Nous terminons ce bref survol des exceptions génériques par un exemple dont nous pensons qu'il reflète la "philosophie" de leur emploi.

Exemple 4.11 *Soit t un type totalement ordonné par une relation d'ordre rel, et soit s une séquence dont les éléments sont de sorte t. Lorsque cette séquence n'est pas vide, nous nous proposons, sans modifier l'ordre d'apparition des éléments, de la partitionner en chaînes ordonnées par rel :*

$$x_1\ rel\ \ldots\ rel\ x_i, x_{i+1}\ rel\ \ldots\ rel\ x_j, \ldots\ x_n$$
$$avec\ \neg(x_i\ rel\ x_{i+1})$$

et de retourner la dernière de ces sous-séquences. Lors du parcours de la séquence s, si deux éléments adjacents x_k et x_{k+1} ne vérifient pas $x_k\ rel\ x_{k+1}$, la construction de la séquence résultat est abandonnée par déclenchement du terme d'exception "$!fail(s_0)$" — où s_0 est une variable de sorte $Seq[t]$ représentant la sous-séquence dont x_{k+1} est le premier élément — et une nouvelle tentative de construction du résultat est appliquée à s_0.

```
enr req Total-Order[t / rel, eq]
  opns  f      : Seq[t]       →  Seq[t]
  priv  f0     : t × Seq[t]   →  Seq[t]
  exc   !fail  : Seq[t]
  vars  x, y   : t
        s, s0  : Seq[t]
  rules
    <>          f(nil)   ==>  nil
    <>      f(x <+ s)    ==>  rec  f0(x, s)
                                   when !fail(s0) then f(s0)
                              endrec
    <>        f0(x, nil) ==>  [x]
    <>   f0(x, y <+ s)   ==>  if x rel y then x <+ f0(y, s) else !fail(y <+ s)
                              endif
endenr
```

$$
\begin{array}{rcl}
f.INCNAT([1,2,1,4]) & \longrightarrow & [1,4] \\
f.DECNAT([1,2,1,4]) & \longrightarrow & [4]
\end{array}
$$

4.5.4.3 Conditions et termes booléens

Comme nous l'avons déjà remarqué, la condition sur les termes booléens utilisés en tant que prémisses est nécessaire pour manipuler des équations conditionnelles et des contraintes de sortes. Une autre solution pourrait être de considérer une classe d'atomes dont la négation ne pose pas de problème, afin de pouvoir utiliser cette dernière pour les contraintes de sortes. Ce problème ne se pose que pour les règles conditionnelles et les contraintes de sortes. Signalons qu'il est possible de simuler un schéma *if-then-else* avec exception (avec la convention selon laquelle si la condition déclenche une exception, le résultat de l'expression est cette exception) à l'aide d'un opérateur caché :

$$
\begin{array}{l}
\textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \\
\textbf{endif}
\end{array}
$$

Ordonnons l'ensemble des variables libres de cette expression ($var(c) \cup var(e_1) \cup var(e_2)$) en une famille $(x_1, \ldots, x_q)$, les sortes respectives étant $s_1, \ldots, s_q$. Ce terme est alors remplacé par le terme $\varpi(c, x_1, \ldots, x_q)$, avec :

$$\varpi : \widehat{Bool} \times s_1 \times \cdots \times s_q \rightarrow \widehat{s}$$

opérateur caché défini par :

$$
\begin{array}{rcl}
\varpi(true, x_1, \ldots, x_q) & == & e_1 \\
\varpi(false, x_1, \ldots, x_q) & == & e_2 \\
\varpi(\phi^1_{\widehat{Bool}}(\tau_{1,1}, \ldots, \tau_{1,\varpi_1}), x_1, \ldots, x_q) & == & \phi^1_{\widehat{s}}(\tau_{1,1}, \ldots, \tau_{1,\varpi_1}) \\
 & \vdots & \\
\varpi(\phi^n_{\widehat{Bool}}(\tau_{n,1}, \ldots, \tau_{1,\varpi_n}), x_1, \ldots, x_q) & == & \phi^n_{\widehat{s}}(\tau_{n,1}, \ldots, \tau_{1,\varpi_n})
\end{array}
$$

4.5.4.4 Exceptions et sémantique de Gert Smolka

Si nous avons préféré donner aux exceptions une sémantique "à la Goguen et Meseguer", c'est pour faire davantage apparaître la construction des deux supports pour chaque sorte introduite par l'utilisateur. Il est immédiat qu'il est possible d'étudier les présentations avec exceptions suivant la sémantique de Gert Smolka. Là encore, nous pouvons nous ramener aux définitions que nous avons rappelées au chapitre 0 en ne considérant pas la surcharge de FP2, surcharge qui, ainsi que nous l'avons vu dans la première partie, n'est qu'une facilité purement syntaxique.

4.5.5 Conclusion de l'étude sémantique

Il est certain que la présentation avec sortes ordonnées obtenue à partir d'une présentation avec exceptions peut apparaître comme compliquée, et en tous cas bien moins naturelle qu'une présentation avec sortes ordonnées spécifiée directement. Remarquons toutefois qu'au niveau du traitement des constructeurs, elle correspond bien au cahier des charges intuitif : certains constructeurs partiels peuvent former des termes qui se réécrivent en des termes exceptionnels, et les autres valeurs qu'ils engendrent sont les termes non exceptionnels de la sorte considérée. Malgré sa complexité, ce cadre nous semble bon pour l'étude des exceptions :

- toutes les fonctions considérées sont totales,
- en l'absence d'exceptions, il coïncide avec le formalisme classique,
- il permet, pour les présentations avec exceptions, d'utiliser beaucoup de résultats dégagés dans le cadre des sortes ordonnées, la plupart des "bonnes propriétés" requises étant satisfaite par les présentations avec sortes ordonnées correspondantes (régularité, cohérence, décroissance des sortes),
- il inclut le cas générique, ce dernier point nous semblant décisif, surtout dans la mesure où, à notre connaissance, aucune des sémantiques proposées jusqu'alors ne permettait de traiter ce cas.

4.6 Traitement local des exceptions dans les processus FP2

En programmation parallèle et distribuée, les principaux problèmes posés par les exceptions sont leur possible occurrence pendant une communication, ce qui peut mettre en cause l'*atomicité* de cette communication, et le maintien de la synchronisation entre processus après un traitement d'exception. Il s'agit de sujets très difficiles, qui dépassent le cadre de cet ouvrage — nous renvoyons à [66] pour la vérification de synchronisation. Nous allons nous contenter ici d'un objectif plus modeste : une proposition d'un traitement local des exceptions à l'intérieur d'un processus FP2.

4.6.1 Nécessité d'intégrer un mécanisme d'exceptions

Nous allons donc maintenant, dans le cadre de FP2, nous intéresser à l'occurrence d'exceptions non récupérées lors de l'exécution d'un processus. Rappelons que les constructeurs d'états d'un processus peuvent être considérés comme des constructeurs d'un type "état de ce processus". Si un déclenchement a lieu lors de l'évaluation d'un tel constructeur (dans une postcondition), la règle d'absorption va par conséquent répercuter l'exception au niveau de toute la postcondition. Tout réseau de processus se ramenant à une présentation de processus élémentaire, il s'ensuit que durant

la simulation d'un réseau, en cas de déclenchement dans l'évaluation d'un terme fonctionnel, le réseau *tout entier* se bloque[66]. De même, une exception survenant lors de l'évaluation d'un événement bloque le processus alors qu'il tente d'appliquer une règle de transition. Ce que nous voulons apporter à FP2, au moyen des exceptions, c'est la possibilité de bloquer *localement* un processus, les autres processus pouvant continuer leur exécution dans la mesure où leurs communications sont satisfaites.

Permettre le blocage partiel à l'intérieur d'un processus étant peu envisageable, il est nécessaire de *généraliser* le traitement des réseaux, de ne plus les comprendre comme équivalents à des processus élémentaires, et d'introduire une nouvelle représentation des états des processus sous forme de **vecteurs**, afin de conserver de façon indépendante l'information sur l'état de chaque processus appartenant au réseau. Tout processus élémentaire est dès lors compris comme un vecteur de processus de longueur 1. C'est alors un vecteur d'état qui caractérise l'état d'un réseau et il devient possible que certaines de ses composantes admettent un paramètre exception, auquel cas les processus correspondants sont bloqués. La forme des règles est modifiée : une précondition est un sous-vecteur de prédicats (les états des processus participant à la transition) et la postcondition un sous-vecteur de même domaine. On peut également signaler que cela conduit à la disparition des règles immobiles, introduites au §0.2.2.1. Cette représentation sous forme de vecteurs permet en outre de diminuer le nombre des règles des présentations de processus[67]. Elle a été esquissée dans [159].

4.6.2 Introduction des exceptions — Conséquences

Réexaminons le calcul d'une règle de transition d'un processus FP2. Les cas spécifiques aux processus sont l'apparition d'une exception non récupérée lors de l'évaluation d'une postcondition ou d'un événement.

Il peut être intéressant de pouvoir récupérer une exception déclenchée lors du calcul d'une postcondition. Afin de ne pas trop nous éloigner de la formulation habituelle de FP2, nous conviendrons que la récupération éventuelle doit avoir lieu au cours de la transition qui a déclenché l'exception. Un processus FP2 ne peut donc pas rester bloqué arbitrairement longtemps : soit la postcondition possède comme paramètres des termes non exceptionnels, pouvant éventuellement provenir d'une récupération d'exception, soit son argument est une exception non récupérée, auquel cas le processus est bloqué à jamais. Une proposition serait de pouvoir "coiffer" toute postcondition d'un "récupérateur d'état", à l'instar des termes de la partie fonctionnelle, soit :

rec $P_0(\ldots)$
  when $!exc_1(\tau_{1,1}, \ldots, \tau_{1,\text{arity}(!exc_1)})$ **then** $P_1(\ldots)$
  $\vdots$
  when $!exc_n(\tau_{n,1}, \ldots, \tau_{n,\text{arity}(!exc_n)})$ **then** $P_n(\ldots)$
  otherwise $P_{n+1}(\ldots)$
endrec

La solution qui a été retenue dans [111] consiste en des récupérateurs liés non pas aux termes d'état, mais aux constructeurs d'états.

[66] Bien sûr, le même problème de blocage général se pose si une évaluation fonctionnelle ne termine pas.

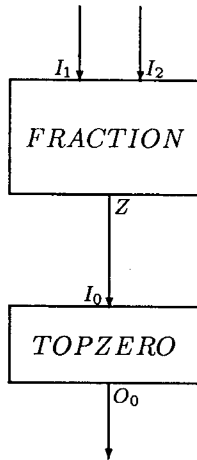
[67] En appliquant les formules données au §0.2.2.1, on obtient que pour deux présentations de processus P_1 et P_2 comportant respectivement p_1 et p_2 règles de transition, respectivement π_1 et π_2 constructeurs d'états, la présentation du processus $P_1 \parallel P_2$ admet $p_1.p_2 + p_1.\pi_2 + \pi_1.p_2$ règles de transition, π_1 et π_2 correspondant, pour chaque processus, aux nombres de règles immobiles à générer. Si l'on adopte la représentation sous forme de vecteurs, le nombre de règles de transition de $P_1 \parallel P_2$ descend à $p_1.p_2 + p_1 + p_2$.

when !*exc* **in** P ==> $P_0(\ldots)$

(Elle correspond à la mise en facteur commun d'un unique récupérateur pour toutes les règles de transition dont les postconditions sont préfixées par P.)

Étudions maintenant, à travers l'exemple suivant, les exceptions non récupérées, qui sont retournées lors de l'évaluation d'un événement.

Exemple 4.12 *Considérons les deux processus suivants : $FRACTION$, qui construit un rationnel à partir de deux entiers, et $TOPZERO$, qui renvoie 0 par son connecteur O_0 dès qu'un rationnel arrive par le connecteur I_0.*




```
process FRACTION
   connectors  I1, I2  :  Nat
               Z       :  PRat
   states      A       :  –
   vars        i, j    :  Nat
   rules
      <>                                 ==>  A
      <>  A  :  I1(i) I2(j) Z(i // j)    ==>  A
endprocess

process TOPZERO
   connectors  I0, O0  :  Nat
   states      A       :  –
   vars        x       :  PRat
   rules
      <>                          ==>  A
      <>  A  :  I0(x) O0(0)       ==>  A
endprocess
```

Construisons le processus $FRACTION \parallel TOPZERO + Z.I_0$. On voit bien que rien n'est prévu pour le processus $TOPZERO$ s'il reçoit un terme exceptionnel par I_0. S'il n'est pas possible de faire filtrer des termes exceptionnels par des variables, et si l'on ne prévoit pas explicitement l'*entrée* de termes exceptionnels, on comprendra aisément que les communications dont le message est une exception soient *refusées.*

Une conséquence importante est que l'occurrence possible d'exceptions non récupérées dans les événements rend nécessaire l'évaluation des messages, même si a priori leur valeur est inutile. Montrons-le en considérant $FRACTION \parallel TOPZERO + Z.I_0 - Z - I_0$.

En appliquant la méthode esquissée au §0.2.2.1, on obtiendrait la présentation de processus possédant les règles suivantes :

$$\begin{array}{rcrl} & & \texttt{==>} & A.A \\ A.A & : & I_1(i)\, I_2(j)\, O_0(0) \quad \texttt{==>} & A.A \end{array}$$

autrement dit un processus qui renvoie 0 par O_0 chaque fois que deux entiers se présentent sur I_1 et I_2. La transition est possible pour $j = 0$!! Ceci est à rapprocher de la remarque du §4.2.6 : on ne peut se dispenser d'évaluer un message que si l'on est sûr qu'il ne retourne pas une exception.

Note On peut rattacher cette obligation d'évaluer un message afin de s'assurer que ce n'est pas un terme d'exception aux *gardes* qui sont ajoutées lors d'une connexion, pour traiter les messages avec opérateurs non constructeurs (cf. §0.2.2.2 et [159, 160]).

Signalons pour clore cette proposition d'extension que ce problème des opérateurs partiels se pose avec encore plus d'acuité dans le cadre d'une implantation parallèle, surtout en présence d'évaluations qui ne terminent pas : à ce sujet, nous renvoyons à [155, 144].

4.7 Implantation des exceptions

Nous commençons par un bref rappel des échappements de Lisp. Nous pourrons ainsi remarquer que ce mécanisme est très proche des exceptions. Nous montrerons ensuite comment les échappements sont utilisés pour la compilation des exceptions de FP2.

4.7.1 Échappements

Nous allons utiliser, comme au chapitre 2, les conventions de Common Lisp (sur ce point, les différences entre les diverses réalisations de Lisp sont très minimes). On *pose* une expression d'**échappement** *TAG* au moyen d'une forme :

$$\Psi = (\texttt{catch}\ TAG\ F_1 \ldots F_n)$$

- Si au cours de l'évaluation de $F_1, \ldots, F_n$, on rencontre une forme :

  $$(\texttt{throw}\ TAG'\ F'_1 \ldots F'_n)$$

  avec `(eq (eval` TAG`) (eval` TAG'`))` vrai, alors le résultat de Ψ est le résultat de l'évaluation de `(progn` $F'_1 \ldots F'_n$`)` ;

- sinon le résultat est la valeur retournée par l'évaluation de F_n.

On comprend que c'est la possibilité de propagation à travers les appels de fonctions qui rend ce mécanisme voisin des exceptions. Nous donnons ci-après un exemple d'utilisation des échappements.

Exemple 4.13 *Calcul d'une substitution réalisant le filtrage d'un terme v par un terme linéaire u. Si* `(linear-filter` *u v*`)` *rend* `nil`, *c'est que u ne filtre pas v. Sinon, le résultat est une liste à un élément, qui est le filtre de u vers v, représenté par une liste de doublets. Les termes u et v sont représentés par des listes, suivant la forme préfixée usuelle en Lisp —* `(f` x_1 `...` x_n`)` *—, et nous supposons l'existence d'un prédicat* `is-variable`, *qui reconnaît les variables (*`x` *et* `y` *dans le dernier exemple de filtre).*

```
(defun linear-filter (u v)
  (catch 'filter (list (rec-linear-filter u v))))

(defun rec-linear-filter (u v)
  (cond ((is-variable u) (list (cons u v)))
```

```
        ((eq (car u) (car v)) (mapcan #'rec-linear-filter (cdr u) (cdr v)))
        (t (throw 'filter nil))))

(linear-filter '(f) '(g))
= nil

(linear-filter '(f) '(f))
= (nil)

(linear-filter '(h x y) '(h a (h b c)))
(((x . a) (y h b c)))
```

Comme on peut le remarquer à travers cet exemple, l'utilisation d'échappements dans une fonction récursive conduit dans la plupart des cas à dissocier les appels récursifs de l'appel principal, qui sert juste à poser les échappements nécessaires. En effet, un échappement ne peut se propager — à la manière d'une exception — qu'à l'intérieur d'une forme où il est posé. On voit que l'absence de propagation par défaut rend ce mécanisme moins souple que celui des exceptions. Signalons que les échappements ont été également intégrés au langage GRAAL [7] (inspiré de FP [4]) de Patrick Bellot.

4.7.2 Implantation des exceptions de FP2 en Common Lisp

Il ressort du §4.5 qu'il est possible d'implanter les présentations avec exceptions comme on implante les présentations avec sortes ordonnées. Nénomoins, l'implantation réalisée "colle" davantage au comportement que nous avons décrit opérationnellement. Elle ne considère pas le système de réécriture équivalent à la règle d'absorption, mais simule directement cette dernière à l'aide du mécanisme d'échappement de Common Lisp. Si le résultat final d'une évaluation est une exception, la conservation de la sorte du résultat (cf. §2.1.1) permet d'indiquer, en même temps que le résultat de l'évaluation du terme d'exception, quelle était la sorte attendue pour ce résultat.

Chaque fois que nous avons à évaluer une expression Lisp provenant d'une compilation d'un terme FP2, nous posons l'échappement "`*fp2-tag-eval*`" ("`*fp2-tag-eval*`" est donc une variable globale de l'évaluateur de FP2). Ainsi, nous pouvons représenter tout déclenchement d'exception par une sortie de cet échappement. Pour un terme de la forme $!exc_0$, le code généré est :

```
(throw *fp2-tag-eval* 'exc0)
```

Nous avons assuré qu'en l'absence de récupération, une sortie de cet échappement sera interceptée par le *top-level* de FP2. Étudions maintenant la compilation d'un récupérateur.

rec t_0
  when $!exc_1$ **then** t_1
  ⋮
  when $!exc_n$ **then** t_n
endrec

Le comportement que nous voulons simuler est : si t_0 n'est pas un terme d'exception, on retourne t_0, sinon on examine le récupérateur. Dans ce cas, si l'exception n'y figure pas, on la propage de nouveau. Soient `BLOCK-NAME` et `X` des symboles qui ne sont pas déjà utilisés dans l'environnement. Le code suivant :

```
(block BLOCK-NAME
  (let ((X (catch *fp2-tag-eval* (return-from BLOCK-NAME t0))))
    (case X
      (exc1 t1)
        :
      (excn tn)
      (otherwise (throw *fp2-tag-eval* X)))))
```

traduit exactement ce comportement[68]. Dans le cas d'un récupérateur comportant une clause "**otherwise**", on peut optimiser le code en supprimant la forme `let` :

```
(block BLOCK-NAME
  (case (catch *fp2-tag-eval* (return-from BLOCK-NAME t0))
     (exc1 t1)
       :
     (excn tn)
     (otherwise tn+1)))
```

Nous remarquerons que *nulle part*, nous n'avons utilisé de test de sorte ou de condition *is-exception*. Autrement dit, un mécanisme d'échappement permet d'implanter les exceptions sans test de sorte.

Cette méthode est bien sûr généralisable aux exceptions avec paramètres. Un terme de la forme $!exc_0(x_1, \ldots, x_n)$ peut être compilé en `(throw *fp2-tag-eval* (list 'exc0 x1 ...xn))`. Il est alors nécessaire de modifier également la compilation des récupérateurs pour tenir compte du filtrage des arguments des exceptions.

À noter qu'une autre solution pourrait être de représenter chaque exception par un échappement différent de Lisp. Ainsi le déclenchement d'une exception remonterait droit jusqu'à son récupérateur, au lieu de cascader par étapes intermédiaires dans le cas où l'exception doit traverser des récupérateurs dans lesquels elle ne figure pas. Cette méthode est a priori séduisante, mais exige l'empilement de nombreux échappements avant toute évaluation d'expression (dans la boucle du *top-level* de FP2, par exemple). En fait, il faut soit empiler tous les échappements connus dans l'environnement, soit connaître ceux qui sont indispensables, par exemple au moyen de techniques d'interprétation abstraite.

[68]`block` et `return-from` sont des formes analogues à `catch` et `exit`, mais leur portée est strictement *lexicale*, c'est-à-dire que l'on ne peut pas empiler un appel de fonction entre `block` et `return-from`. Par contre, leur utilisation est moins coûteuse.

Partie III

Chapitre 5

Transformation de programmes

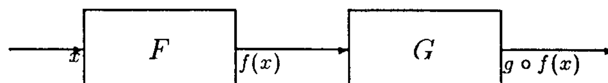
Nous allons nous intéresser maintenant à un autre aspect, non encore implanté, du projet FP2 : la transformation de programmes fonctionnels récursifs en programmes parallèles. Par certaines considérations, nous allons, dans ce chapitre, nous écarter quelque peu du domaine des spécifications algébriques et nous situer parfois dans une perspective plus opérationnelle, nous attachant, par exemple, à certains problèmes liés à la récursivité des opérateurs. Ce que nous cherchons, c'est dégager des possibilités et des principes de transformations en utilisant les deux parties, fonctionnelle et parallèle, du formalisme de FP2. Du point de vue de la partie fonctionnelle, nous mettrons aussi en évidence une utilisation des outils de la généricité pour la formalisation des schémas de programmes que nous désirons transformer.

5.1 Les idées de base

5.1.1 Introduction informelle

Un opérateur f, de profil $t \rightarrow t_0$, peut être vu comme un processus acceptant une donnée x de sorte t et fournissant en sortie le résultat $f(x)$.

À l'appui de cette démarche, remarquons que la composition de deux opérateurs peut alors être rapprochée d'une mise en parallèle des deux processus qui représentent les opérateurs, suivie d'une connexion :



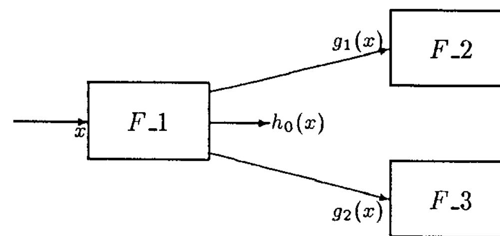
Considérons une fonction f dont la spécification fait apparaître deux appels récursifs, c'est-à-dire :

$$f(x) \quad ==> \quad \textbf{if } c(x) \textbf{ then } h_0(x) \textbf{ else } h(f(g_1(x)), f(g_2(x))) \textbf{ endif} \tag{5.1}$$

Exemple 5.1 *Fonction de Fibonacci :*

$$\left.\begin{array}{rcl} c(x) & ==> & x = 1 \\ h_0(x) & ==> & 1 \\ h(x,y) & ==> & x + y \\ g_1(x) & ==> & pred(x) \\ g_2(x) & ==> & pred(pred(x)) \end{array}\right\} \quad (5.2)$$

L'absence de contrôle explicite et d'effets de bord, caractéristiques des langages fonctionnels, permet d'envisager l'évaluation en parallèle des deux arguments de h, et ce par des processus similaires à celui qui reçoit l'argument de f. Apparaît ainsi l'idée d'une disposition arborescente :



L'application à FP2 de ce principe de transformation a d'abord été présentée en [37] dans un cas particulier : le calcul d'un plus petit unificateur (*most general unifier*) de deux termes construits avec des opérateurs dyadiques. En effet :

$$mgu(\overline{f_0(u_1,u_2)}, \overline{f_0(v_1,v_2)}) = solve\text{-}clash(mgu(u_1,v_1), mgu(u_2,v_2))$$

où *solve-clash* résout les conflits de variables des deux substitutions qu'elle reçoit en paramètres et retourne en cas de succès la substitution globale (cf. §3.5.1).

Toujours dans le cas particulier de l'unification, la transformation entière a été développée dans [104] : nos objectifs étaient, d'une part, de faire ressortir l'expressivité de FP2, d'autre part, de montrer, pour le problème de l'unification, ce qui *doit* s'exécuter séquentiellement, et ce qui *peut* s'exécuter parallèlement (rapellons que, du point de vue des difficultés et de la complexité, la parallélisation de l'unification a été étudiée dans [54, 176]). Nous avons ensuite présenté le cas général de cette transformation dans [106]. Nous reprenons ici cette approche en montrant qu'une large partie des hypothèses peut être exprimée au moyen de la généricité. Les processus obtenus sont génériques, directement adaptables à toute fonction qui satisfait les hypothèses de départ. Si ce chapitre est avant tout un exposé de notre méthode de transformation, c'est aussi un exemple ambitieux d'utilisation de la généricité dans le cadre des processus.

5.1.2 Cadre de l'étude

Nous allons ici évoquer précisément ce que représente la notion de transformation de programmes fonctionnels en processus FP2. De même que l'on ne peut donner un sens aux transformations entre programmes fonctionnels que par la donnée d'une équivalence entre deux expressions fonctionnelles, il nous est nécessaire de définir une équivalence entre calcul fonctionnel et calcul effectué au moyen d'un système de transition.

Définition 5.2 *Un opérateur f, de profil $s_1 \times \cdots \times s_n \rightarrow s$, et de propriété exigée p, est* **simulable** *par la présentation de processus P, qui comporte les connecteurs $C_1, \ldots, C_n, C_{n+1}$, de sortes respectives*

$s_1, \ldots, s_n, s$, et qui admet également p comme propriété exigée, si, pour toute instanciation fermée de p, le système de transition que représente P est tel que l'acceptation de n termes $\underline{x_1, \ldots, x_n}$ par les connecteurs $C_1, \ldots, C_n$ est suivie, au bout d'un temps fini, par l'émission du terme $\overline{f(x_1, \ldots, x_n)}$ sur le connecteur C_{n+1}.

Pour évoquer la *faisabilité* de ces transformations de programmes dans le cadre de FP2, et préciser le rôle de la partie générique, on peut établir un second rapprochement entre transformations de programmes fonctionnels, dans le but d'améliorer les performances, et transformations de programmes fonctionnels en processus parallèles. Dans le premier cas, la programmation "à la FP", sans variable, s'est révélée être un bon formalisme, en permettant d'exprimer des équivalences entre expressions de fonctions. Dès lors que la plupart des formes fonctionnelles du langage FP s'expriment au moyen d'opérateurs génériques[69], on peut, en s'inspirant des travaux accomplis dans le cadre de FP, exprimer des lois entre opérateurs génériques, par exemple :

$$alpha[g] \circ alpha[f] = alpha[g \circ f]$$

(où "o" noté ici sous la forme traditionnelle infixée est également un opérateur générique (cf. annexe B, exemple B.7). On peut donc utiliser cette formule de gauche à droite, pour éviter la construction d'une séquence intermédiaire résultat de *alpha*$[f]$. De nombreux résultats de ce genre, utilisables en LPG et en FP2, sont exposés dans [9]. L'idée est que ces formules de transformation peuvent être décrites par des règles de réécriture qui s'appliquent à des expressions d'opérateurs génériques. Pour l'instant, ce qui manque à la mise en œuvre d'un tel atelier de transformations, ce sont d'une part un méta-langage permettant de construire, de manipuler des termes et des définitions d'opérateurs, et d'autre part une interface permettant à l'utilisateur de guider les transformations.

Afin qu'un atelier de transformations, utilisant des principes analogues, puisse traiter les transformations de fonctions en processus, il devient alors nécessaire que le méta-langage intègre la possibilité de générer des présentations de processus, la généricité fournissant la généralisation indispensable sur les signatures. Nous allons maintenant construire un réseau de processus pouvant simuler un schéma de double récursivité analogue à la règle (5.1). Nous précisons d'abord nos hypothèses, puis les spécifications algébriques dont nous aurons besoin.

5.2 Hypothèses — Définitions fonctionnelles nécessaires

5.2.1 Les hypothèses

Soit f un opérateur défini sur t et à valeurs dans t_0, dont la spécification peut s'écrire selon un schéma analogue à (5.1). Nous faisons de plus les hypothèses suivantes :

(i) h est associative et commutative,

(ii) c, h_0, h, g_1, g_2 sont telles que $(\forall x \in t)$, l'évaluation de $f(x)$ termine.

Traduire que f est reliée à c, h_0, h, g_1, g_2 d'après l'équation (5.1) et l'hypothèse (i) peut être réalisé comme suit à l'aide d'une propriété :

[69] Voir l'exemple 3.2 et l'annexe B (exemple B.7). Il est également possible de simuler entièrement FP à l'aide de règles de réécriture en définissant deux types correspondant respectivement aux objets et opérateurs de FP [103].

```
prop Double-Rec[t, t0 / f, c, h0, h, g1, g2]
  opns  f, h0   : t        → t0
        c       : t        → Bool
        h       : t0 × t0  → t0
        g1, g2  : t        → t
  vars  x       : t
  rules
    <>  f(x)  ==>  if c(x) then h0(x) else h(f(g1(x)), f(g2(x)))
                   endif
  includes  Associativity[t0 / h], Commutativity[t0 / h]
endprop
```

Si l'on admet les règles de réécriture dans les propriétés et l'héritage de telles propriétés dans les définitions d'opérateurs (cf. §1.2.6, note), alors une instanciation possible définissant la fonction de Fibonacci est :

```
enr
  opns      fib : Nat → Nat
  inherits  Double-Rec[Nat, Nat / fib, curry[≤, 1], cst[1], +, pred, o[pred, pred]]
endenr
```

Nous avons traduit sous une forme entièrement "à la FP" les définitions du système d'équations (5.2) — la spécification complète des opérateurs génériques *cst*, o et *curry* est donnée à l'annexe B (exemple B.7).

On peut également, et c'est la solution que nous adoptons, concevoir un opérateur générique rec^2 simulant un schéma de double récursivité, que l'on définit comme suit :

```
prop For-Double-Rec[t, t0 / c, h0, h, g1, g2]
  opns  h0      :    t      → t0
        c       :    t      → Bool
        h       :  t0 × t0  → t0
        g1, g2  :    t      → t
  includes  Associativity[t0 / h], Commutativity[t0 / h]
endprop
```

```
enr req For-Double-Rec[t, t0 / c, h0, h, g1, g2]
  opns  rec²  : t → t0
  vars  x     : t
  rules
    <>  rec²(x)  ==>  if c(x) then h0(x) else h(rec²(g1(x)), rec²(g2(x)))
                      endif
endenr
```

Alors :

$$fib(x) ==> rec^2[curry[\leqslant, 1], cst[1], +, pred, \circ[pred, pred]](x)$$

définition qu'il est possible d'écrire sans utiliser de variable :

$$fib = rec^2[curry[\leqslant, 1], cst[1], +, pred, \circ[pred, pred]]$$

Note L'hypothèse de départ — double appel récursif dans la définition de l'opérateur — n'est pas aussi restrictive qu'il n'y paraît. En effet, dans [50], et surtout dans [36], ont été développées des méthodes de

transformation de définitions avec simple appel récursif en des définitions avec double appel récursif. À titre d'exemple, nous reproduisons ci-dessous le résultat obtenu dans [36] pour la fonction factorielle :

$$\begin{array}{rcl} n! & = & dfac(n,0) \\ dfac(n,p) & = & \textbf{if } \; n \leqslant p \textbf{ then } 1 \\ & & \quad \textbf{elsif } \; pred(n) \leqslant p \textbf{ then } n \\ & & \quad \textbf{else } \; \textbf{let } \; q :: \lfloor \frac{n+p}{2} \rfloor \\ & & \qquad\qquad \textbf{in } dfac(n,q) * dfac(q,r) \\ & & \qquad \textbf{endlet} \\ & & \textbf{endif} \end{array}$$

5.2.2 Type "zéro ou un"

Nous aurons par ailleurs besoin de représenter "zéro ou un élément de sorte t". Plutôt que d'utiliser des séquences dont la longueur n'excède pas 1, nous avons préféré introduire un nouveau type générique :

type *ZeroOrOne* **req** *Ftype*[*t* /]
  cons *canonic* : *t*
    nil : –
endtype

On peut comprendre "*ZeroOrOne*[*t*]" comme l'ajout au type t d'un élément minimal. Nous terminons les définitions fonctionnelles de ce chapitre en montrant comment étendre une fonction de profil $t \times t \rightarrow t$ en une fonction de profil $ZeroOrOne[t] \times ZeroOrOne[t] \rightarrow ZeroOrOne[t]$.

prop *Dyadic-Op*[*t* / *h*]
  opns $h \;:\; t \times t \;\rightarrow\; t$
endprop

enr req *Dyadic-Op*[*t* / *h*]
  opns $ext \;:\; ZeroOrOne[t] \times ZeroOrOne[t] \;\rightarrow\; ZeroOrOne[t]$
  vars $x, x_0 \;:\; t$
  rules
    <> $ext(canonic(x), canonic(x_0))$ ==> $canonic(h(x, x_0))$
    <> $ext(canonic(x), nil)$ ==> $canonic(x)$
    <> $ext(nil, canonic(x_0))$ ==> $canonic(x_0)$
    <> $ext(nil, nil)$ ==> nil
endenr

Proposition 5.3

- *Si h est associative (resp. commutative), ext*[*h*] *est également associative (resp. commutative).*
- *ext*[*h*] *admet un élément neutre qui est nil.*

On peut exprimer cette proposition par les modèles génériques suivants :

model req *Associativity*[*t* / *h*] **is**
  Associativity[*ZeroOrOne*[*t*] / *ext*[*h*]]
endmodel

model req *Commutativity*[*t* / *h*] **is**
  Commtutativity[*ZeroOrOne*[*t*] / *ext*[*h*]]
endmodel

model req *Dyadic-Op*[t / h] **is** *Neutral*[*ZeroOrOne*[t] / $ext[h], nil$]
endmodel

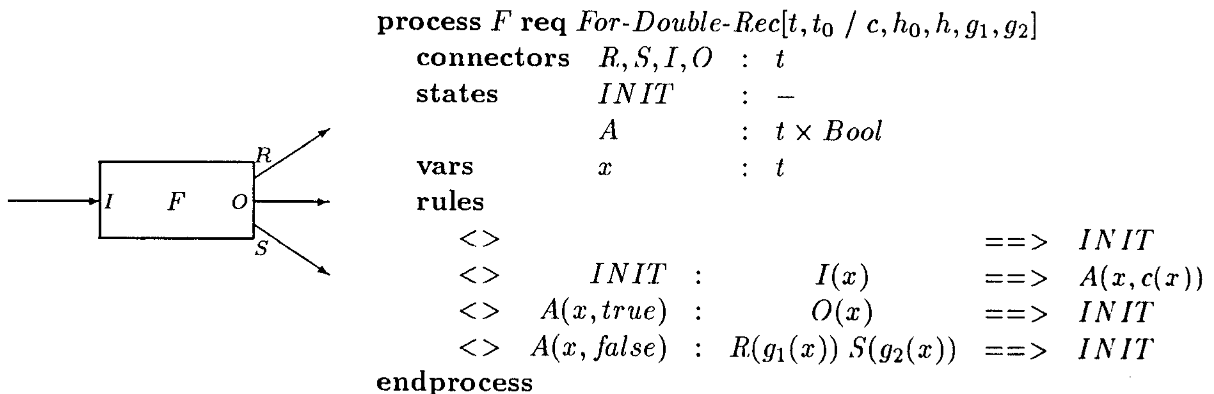
Remarque L'extension de h à $ext[h]$ revient à ajouter un élément neutre à h. Nous verrons dans notre exemple, que *nil*, l'élément "parasite" que nous avons rajouté, est totalement invisible de l'utilisateur qui ne voit que les entrées et les sorties du processus simulant la fonction f.

Nous signalons une autre utilisation du type *ZeroOrOne*[t] dans le cadre des processus : lire en entrée d'un processus des éléments d'une sorte t, un à un, puis un signal de fin d'entrées. Dans ce dernier cas, on a bel et bien besoin de reconnaître les termes de sorte t et un terme supplémentaire, dont la sorte n'est pas t.

5.3 Transformation : la démarche

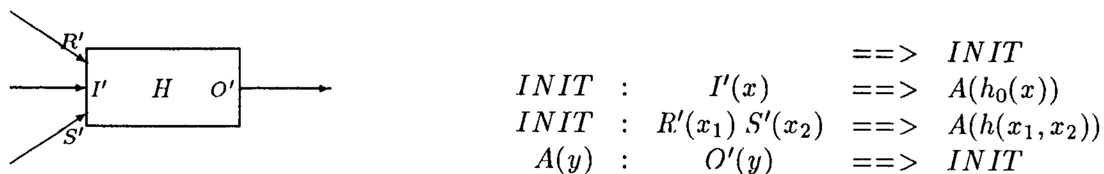
5.3.1 Parallélisme en largeur : le réseau obtenu

Reprenant l'idée esquissée au §5.1.1, nous allons donner la présentation du processus F, paramétrée génériquement par la propriété *For-Double-Rec*. Il accepte une entrée x de sorte t. Si $c(x)$ est vrai (pas d'appel récursif), F renvoie x via le connecteur O, sinon deux valeurs correspondant aux appels récursifs à simuler sont envoyés via les connecteurs R et S. La spécification de F est :



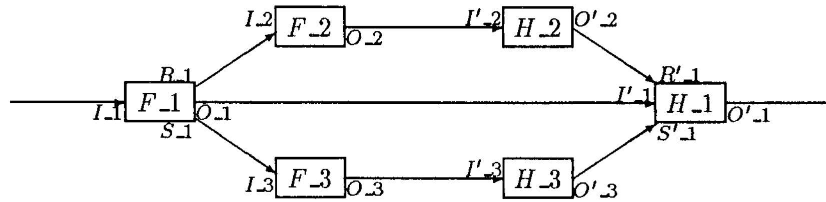

```
process F req For-Double-Rec[t, t0 / c, h0, h, g1, g2]
  connectors  R, S, I, O  : t
  states      INIT        : –
              A           : t × Bool
  vars        x           : t
  rules
    <>                                          ==> INIT
    <>     INIT        :   I(x)                 ==> A(x, c(x))
    <>     A(x, true)  :   O(x)                 ==> INIT
    <>     A(x, false) :   R(g1(x)) S(g2(x))    ==> INIT
endprocess
```

Introduisons à présent le "miroir" d'un processus F : il doit calculer l'image par h_0 d'une entrée présente sur le connecteur I' ou l'image par h de deux entrées présentes sur R' et S'. Comme celle de F, la présentation de H est paramétrée génériquement par la propriété *For-Double-Rec*. Les règles de H sont :




```
                                    ==> INIT
INIT  :   I'(x)                     ==> A(h0(x))
INIT  :   R'(x1) S'(x2)             ==> A(h(x1, x2))
A(y)  :   O'(y)                     ==> INIT
```

Nous assemblons les processus F et H en un réseau sous forme de losange.



(5.3)

$$F_1 \parallel F_2 \parallel F_3 + R_1.I_2 + S_1.I_3 \parallel$$
$$H_1 \parallel H_2 \parallel H_3 + O'_2.R'_1 + O'_3.S'_1 + O_1.I'_1 + O_2.I'_2 + O_3.S'_3 -$$
⟨⟨abstraction des connecteurs encore visibles⟩⟩

Remarque Dans l'expression précédente, nous avons écrit "F_i" pour "*F_i.For-Double-Rec*[.../...]". Pour ne pas alourdir les notations, nous utiliserons cette convention dans la suite du chapitre.

Le réseau représenté en (5.3) peut calculer $f(x)$ pour tout x dont le calcul de l'image par f ne nécessite pas plus d'un double appel récursif. À titre d'exemple, un tel réseau pour la fonction *fib* peut évaluer *fib*(x) pour $x \leqslant 2$.

Il est possible de paramétrer un réseau par sa profondeur[70], donc de choisir celle-ci aussi grande que l'on veut :

process *DIAMOND* [n : *Nat*] **req** *For-Double-Rec*[.../...] **is**
$(\|_{1\leqslant i\leqslant n}\ \{F_i\}) + \cdots \parallel (\|_{1\leqslant i\leqslant n}\ \{H_i\}) + \cdots$
endprocess (5.4)

Nous nous heurtons cependant à une limitation sévère : nous ne pouvons ni manipuler un nombre infini de processus, ni en créer dynamiquement au fur et à mesure de nos besoins. En effet, ce dernier mécanisme ne peut pas être modélisé par la formulation actuelle des règles de transition. Souvenons-nous que définir un opérateur entre présentations de processus, c'est expliquer comment obtenir les connecteurs, états et règles de la présentation du processus résultant. On comprend dès lors que la création dynamique de processus soit impossible, du moins pour l'instant, en FP2[71].

Par conséquent, nous devons quoi qu'il en soit examiner le cas des éléments qui nécessitent une pile de récursivité plus importante que la profondeur de notre réseau. À propos de ces éléments, deux remarques vont guider la suite.

[70] Cette facilité — non implantée actuellement et utilisée ici informellement — de paramétrer la *structure* d'un processus ou d'un réseau (facilité qui ne doit pas être confondue avec la paramétrisation au moyen de la généricité) est en fait une *macro* qui est développée à la compilation, lorsque les variables représentant la paramétrisation de la structure ont reçu une valeur.

[71] Philippe Jorrand a récemment proposé l'extension suivante pour l'activation dynamique de processus sur une structure de réseau prédéfinie. Si une règle est de la forme :

: ⟨événement⟩ ==> ⟨postcondition⟩

sans précondition, cela signifie que le processus n'est créé qu'à l'occurrence de l'événement ; la substitution qui valide l'événement est alors appliquée à la postcondition pour obtenir l'état du processus.

De même, une règle de la forme :

⟨précondition⟩ : ⟨événement⟩ ==>

sans postcondition, spécifie que le processus disparaît après application de cette règle.

Par exemple, soient les processus P_1 et P_2 tels que :

1. Il semble judicieux de profiter du maximum de profondeur de l'arbre des $\{F_i\}$, c'est-à-dire qu'il vaut mieux acheminer de proche en proche vers la sortie du processus H_1 les deux termes dont on ne peut simuler directement l'appel récursif de f, et ensuite les renvoyer à l'entrée du processus F_1.

2. Étant donné deux tels termes, les deux évaluations de leurs images par f sont également indépendantes. On peut par conséquent les effectuer elles aussi en parallèle, c'est-à-dire envoyer le premier terme, puis le second dès que F_1 est prêt à accepter une nouvelle entrée, sans nécessairement attendre la fin du calcul du premier.

Tout ceci nous suggère une architecture *pipe-line*. Nous allons à présent clarifier le traitement des termes *trop profonds*, puis étudier précisément le *pipe-line*.

5.3.2 Les termes trop profonds

En accord avec la première des remarques précédentes, les processus $\{H_i\}$ doivent pouvoir transmettre, de proche en proche, une séquence de termes de sorte t. Nous révisons donc la spécification de H et les cas d'entrée se répartissent désormais comme suit :

⟶ il peut arriver un "sous-résultat" (résultat partiel dans l'évaluation de f) sur le connecteur I' : ce sous-résultat est soit un terme spécial traduisant qu'aucun appel de h n'a eu lieu, soit un terme de sorte t_0 correspondant à une valeur prise par la fonction h ;

⟶ sur les connecteurs R' et S', il peut arriver éventuellement un sous-résultat et une séquence (éventuellement vide) de termes de sorte t que l'on doit transmettre.

Pour concilier tous ces objectifs, les messages des connecteurs R', S' et D' seront de sorte :

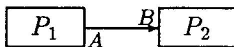
$$CP^2[ZeroOrOne[t_0], Seq[t]]$$

où CP^2 représente le produit cartésien de deux sortes (cf. annexe, exemple B.10). Son utilisation est ici indispensable, un connecteur étant obligatoirement étiqueté par *une seule* sorte (cf. définition 0.89).

La nouvelle spécification du processus H s'obtient en modifiant les sortes des connecteurs comme indiqué plus haut, en remplaçant dans les règles h par $ext[h]$ (que nous noterons plus simplement $\hbar$), et en tenant compte du passage à travers H des séquences de termes de sorte t. Ce qui donne :

$$\begin{array}{rcccl} & & & ==> & INIT \\ INIT & : & I'(x) & ==> & A(canonic(h_0(x)), nil) \\ INIT & : & R'(\langle y_1, s_1\rangle)\ S'(\langle y_2, s_2\rangle) & ==> & A(\hbar(y_1, y_2), s_1 + s_2) \\ A(y,s) & : & O'(\langle y, s\rangle) & ==> & INIT \end{array}$$

Règles de P_1 :



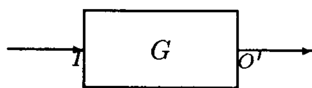
$$\begin{array}{rcccl} & & & ==> & INIT \\ INIT & : & A(0) & ==> & \end{array}$$

Règles de P_2 :

$$\begin{array}{rccl} : & B(i) & ==> & B_0(i) \\ & & \vdots & \end{array}$$

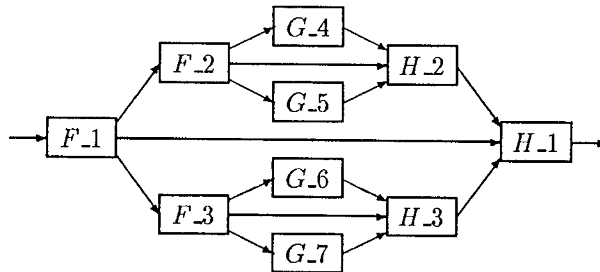
Le processus P_1 active le processus P_2 et disparaît au cours de la même transition.

Introduisons maintenant le processus G, à l'aide duquel nous allons effectuer la jonction entre les processus $\{F_i\}$ et $\{H_i\}$. Ce processus se comporte comme F dans le cas où il n'y a pas d'appel récursif à simuler. Dans le cas contraire, au lieu de simuler les appels récursifs par envoi à des processus F, il rend une séquence de deux termes à acheminer jusqu'à H_1 avant de les renvoyer à F_1.



$$
\begin{array}{rccl}
 & & ==> & INIT \\
INIT & : \quad I(x) & ==> & A(x, c(x)) \\
A(x, true) & : \quad O'(\langle canonic(h_0(x)), nil\rangle) & ==> & INIT \\
A(x, false) & : \quad O'(\langle nil, [g_1(x), g_2(x)]\rangle) & ==> & INIT
\end{array}
$$

Le réseau construit jusqu'à présent est de la forme :



5.3.3 Pipe-line

Une question reste à examiner : comment va-t-on gérer les sorties du processus H_1 et les entrées du processus F_1 ?

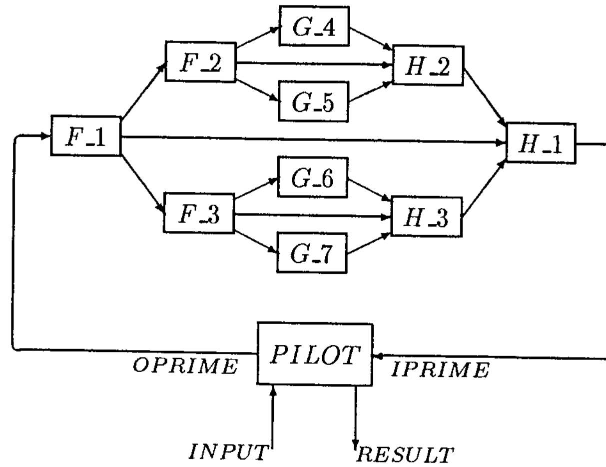
Comme nous l'avons précédemment remarqué, une fois qu'un processus F a rendu sa réponse en ce qui concerne un couple de données, il peut fort bien en traiter un autre. Ceci nous suggère une réalisation en *pipe-line* dirigée par un processus pilote qui :

⟶ reçoit en entrée l'élément dont on veut calculer l'image par f,

⟶ fournit en sortie le résultat du calcul,

⟶ pendant un calcul, gère celui-ci d'après son état $A(i, y, s)$ où :

- ⋆ i peut être compris comme un compteur : c'est le nombre de sous-résultats attendus,
- ⋆ y, de sorte $ZeroOrOne[t_0]$, est le résultat partiel,
- ⋆ s est la séquence de termes de sorte t en attente d'être envoyés à F_1.



Nous donnons et commentons les règles du processus $PILOT$.

$$\begin{array}{rcl} & ==> & INIT \\ INIT \ : \ INPUT(x) & ==> & A(0, nil, [x]) \end{array}$$

Lorsque l'entrée principale est reçue, elle est en attente d'être envoyée au processus F_1.

$$A(0, canonic(y), nil) : RESULT(y) ==> INIT$$

Le calcul est fini lorsqu'on n'attend plus de sous-résultat et lorsqu'il n'y a plus de termes en attent d'envoi à F_1. Le résultat est alors le deuxième argument de A.

$$A(succ(i), y_1, s_1) : IPRIME(\langle y_2, s_2 \rangle) ==> A(i, \hbar(y_1, y_2), s_1 + s_2)$$

Si le compteur est strictement positif, nous attendons au moins un sous-résultat. Si nous le recevor via $IPRIME$, nous le composons par $\hbar$ avec le sous-résultat courant et les deux séquences de term à envoyer à F_1 sont concaténées. Un résultat en moins étant attendu, le compteur diminue de 1.

$$A(i, y, x <+ s) : OPRIME(x) ==> A(succ(i), y, s)$$

Quelque soit la valeur du compteur, si au moins un terme attend d'être envoyé à F_1, on l'envoi et le compteur est augmenté de 1.

5.4 Preuve de correction

Nous rappelons les hypothèses :

- h est associative et commutative,
- c, h_0, h, g_1, g_2 sont telles que $(\forall x \in t)$, l'évaluation de $f(x)$ termine.

La seconde hypothèse assure que toutes les évaluations dont nous allons faire mention retournen effectivement un résultat. Notre preuve comporte trois parties :

(i) Une entrée est acceptée au bout d'un temps fini par un processus *F_i*, et, quand elle est acceptée, au bout d'un temps fini, d'une part, *F_i* est revenu dans son état initial (il a pu accepter une autre entrée), d'autre part, *H_i* est prêt à envoyer la sortie correspondante et à revenir dans son état initial. C'est-à-dire que si l'environnement est prêt à satisfaire une telle communication, l'événement a effectivement lieu.

(ii) Le *pipe-line* termine.

(iii) Le résultat de l'évaluation parallèle de f est correct.

Preuve de (i)

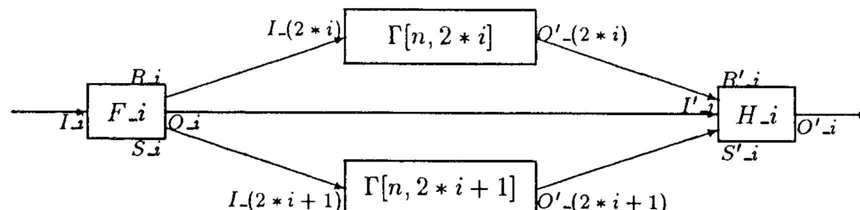
Dans le cadre de FP2, des études ont déjà été consacrées à l'analyse du comportement des processus et aux problèmes de terminaison : citons [146], fondé sur la théorie des fonctions non nœthériennes, qui étudie la terminaison, et [163], qui traite de l'analyse des processus FP2 à l'aide des outils de la logique temporelle (définition de prédicats de possibilité, d'inévitabilité de l'occurrence d'un événement).

Dans l'absolu, il est possible de développer toutes les règles du processus obtenu et de lui appliquer de telles méthodes. Néanmoins, étant donné le nombre de processus manipulés, et par conséquent le nombre de règles que l'on obtiendrait[72], nous préférons donner une preuve moins formelle mais plus simple, par récurrence sur la profondeur du réseau.

Récurrence utilisée

Nous avons vu (cf. processus *DIAMOND* (5.4)) que l'on pouvait paramétrer un réseau par sa profondeur au moyen d'une *macro*. C'est cette facilité que nous allons à nouveau utiliser pour redéfinir notre réseau par récurrence sur sa profondeur. L'idée est qu'un réseau de profondeur 0 est formé par un seul processus *G_i*, i étant un paramètre de la macro spécifiant l'indice. Ensuite, on construit un réseau de profondeur $succ(n)$ à l'aide de deux réseaux de profondeur n et de deux processus F et H.

process $\Gamma[Nat, Nat]$ **req** *For-Double-Rec*[.../...]
vars $n, i : Nat$ -- n est la profondeur, i l'indice de départ pour le renommage
$\Gamma[0, i]$ **is** *G_i*
$\Gamma[succ(n), i]$ **is** $F_i \parallel \Gamma[n, 2*i] + R_i.I_(2*i) \parallel \Gamma[n, 2*i+1] + S_i.I_(2*i+1) \parallel$
$H_i + O'_(2*i).R'_i + O'_(2*i+1).S'_i + O_i.I'_i$
endprocess



[72] Rappelons que pour deux présentations de processus P_1 et P_2 comportant respectivement p_1 et p_2 règles de transition, la représentation plus compacte de $P_1 \parallel P_2$ (sous forme de vecteur) admet malgré tout $p_1.p_2 + p_1 + p_2$ règles de transition (cf. note 67, p. 141). Signalons à titre d'exemple que la programmation sur le prototype de la parallélisation de la fonction *fib* avec une profondeur 2 (et avec les processus d'entrée-sortie, cf. annexe D) donne un processus de 64 règles.

Pour montrer (i), il suffit de montrer que pour tout n, un processus $\Gamma[n,i]$ accepte une entrée au bout d'un temps fini, et, quand il l'a acceptée, au bout d'un temps fini, il est prêt à renvoyer la réponse correspondante.

Histoires des processus F, G et H — Amorce de la récurrence

Commençons par examiner, les histoires possibles des processus G, F et H. Pour le processus G, les deux histoires suivantes, après substitution de la variable x et composition itérée d'un nombre quelconque d'entre elles, permettent de générer toute histoire possible :

$$\begin{array}{lclcl} INIT & \overset{I(x)}{\Longrightarrow} & A(x,true) & \overset{O'(\langle canonic(h_0(x)),nil\rangle)}{\Longrightarrow} & INIT \\ INIT & \overset{I(x)}{\Longrightarrow} & A(x,false) & \overset{O'(\langle nil,[g_1(x),g_2(x)]\rangle)}{\Longrightarrow} & INIT \end{array}$$

De même, pour un processus F :

$$\begin{array}{lclcl} INIT & \overset{I(x)}{\Longrightarrow} & A(x,true) & \overset{O(x)}{\Longrightarrow} & INIT \\ INIT & \overset{I(x)}{\Longrightarrow} & A(x,false) & \overset{R(g_1(x))\ S(g_2(x))}{\Longrightarrow} & INIT \end{array}$$

et pour un processus H :

$$\begin{array}{lclcl} INIT & \overset{I'(x)}{\Longrightarrow} & A(canonic(h_0(x)),nil) & \overset{O'(\langle canonic(h_0(x)),nil\rangle)}{\Longrightarrow} & INIT \\ INIT & \overset{R'(\langle y_1,s_1\rangle)\ S'(\langle y_2,s_2\rangle)}{\Longrightarrow} & A(\hbar(y_1,y_2),s_1+s_2) & \overset{O'(\langle \hbar(y_1,y_2),s_1+s_2\rangle)}{\Longrightarrow} & INIT \end{array}$$

Chacune de ces histoires suit le même schéma : admission de l'entrée (ou des entrées), puis renvoi de la réponse correspondante. Considérons ces processus à l'intérieur d'un réseau. Au niveau global, cela signifie que l'un d'entre eux, s'il accepte une entrée, est prêt à rendre sa réponse correspondante au bout d'un temps fini. D'autres transitions mettant en jeu des processus voisins peuvent bien sûr avoir lieu entre l'acceptation d'une entrée et le renvoi de la réponse correspondante, néanmoins, remarquons également que si l'un des processus F, G, ou H a accepté une entrée, il ne peut en admettre une autre avant d'avoir traité l'entrée courante.

L'examen des règles de G nous montre que la propriété est vraie pour $\Gamma[0,i]$ (quel que soit i). Supposons maintenant qu'elle est vraie pour $\Gamma[n,i]$ et examinons le cas de $\Gamma[succ(n),i]$:

Passage au successeur

À tout moment, le processus H_i est soit dans l'état $INIT$, soit dans un état $A(y,s)$. Si au bout d'un temps fini, l'environnement est prêt à accepter une communication via O'_i, alors la règle :

$$A(y,s) : O'_i(\langle y,s\rangle) ==> INIT$$

et cette règle seulement, est nécessairement appliquée au bout d'un temps fini. Comme, par hypothèse, l'environnement est prêt, au bout d'un temps fini, à satisfaire une telle communication, ceci signifie que H_i finit toujours par repasser dans l'état $INIT$.

Ceci étant posé, supposons que F_i accepte une entrée x. Alors, deux cas se présentent :

$c(x)$ **est vrai.** Dès que l'environnement peut valider les communications de H_i par O'_i, alors d'après le raisonnement précédent, H_i finit par accepter l'entrée x et est alors en mesure de retourner $h_0(x)$ par O'_i.

$c(x)$ **est faux.** Dans ce cas, $g_1(x)$ et $g_2(x)$ sont respectivement envoyés à $\Gamma[n, 2*i]$ et $\Gamma[n, 2*i+1]$, si ceux-ci peuvent les accepter. S'ils ne le peuvent pas, c'est qu'ils traitent une donnée précédente, mais d'après l'hypothèse de récurrence, chacun accepte une entrée au bout d'un temps fini. Alors le processus F_i revient dans son état initial et, toujours d'après l'hypothèse de récurrence, les processus $\Gamma[n, 2*i]$ et $\Gamma[n, 2*i+1]$ sont au bout d'un temps fini prêts à renvoyer leur réponse si l'environnement peut l'accepter. Or les connecteurs $O'_(2*i)$ et $O'_(2*i+1)$ sont reliés respectivement à R'_i et S'_i. Là encore, en utilisant le résultat préliminaire, H_i finit par passer dans l'état $INIT$, attend éventuellement que les deux processus soient prêts, accepte les données et est en mesure de fournir la réponse.

Concluons cette preuve de (i) par les deux remarques suivantes :

1. À tout moment, le pilote peut communiquer avec le processus $\Gamma[n, 1]$, soit via $OPRIME.I_1$, soit via $O'_1.IPRIME$. Par conséquent, lorsque le pilote envoie une donnée à F_1, au bout d'un temps fini, il récupère la réponse venant de H_1.
2. Nous avons démontré qu'il se produit autant d'envois à F_1 que de réponses de H_1. Mais cela ne signifie absolument pas que l'ordre est le même. Par exemple, il est possible que F_1 reçoive une première entrée x_1 telle que $c(x_1)$ soit faux, et une deuxième entrée x_2 telle que $c(x_2)$ soit vrai. x_2 est donc directement envoyé à H_1 et il est tout à fait possible que H_1 accepte et traite x_2 avant les résultats des deux sous-calculs pour x_1.

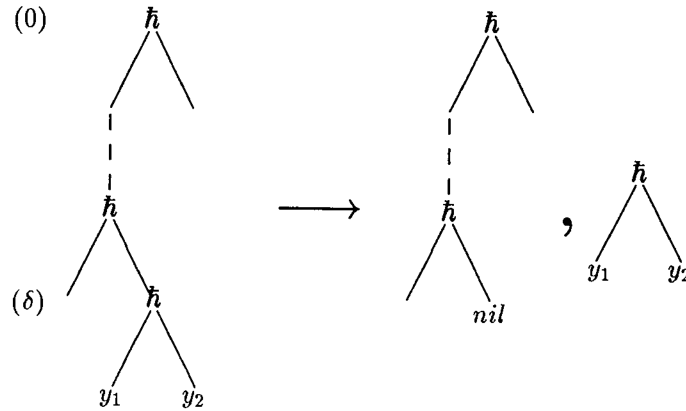
Preuve de (ii)

Si l'évaluation de f termine, pour tout élément de t, le nombre d'appels récursifs de f est fini. Pour un élément quelconque du domaine de f, soit ϕ ce nombre d'appels. Considérons le compteur comme une suite de valeurs et soit κ une valeur quelconque de cette suite.

Le rôle du compteur étant explicité, nous avons toujours $\kappa \leqslant \phi$. Il s'ensuit que (κ) ne peut pas être une suite croissante infinie. Si nous utilisons la preuve de (i), les envois à F_1 qui ont fait "incrémenter" le compteur seront suivis d'autant de réceptions qui le feront "décrémenter". Sa valeur initiale étant 0, sa valeur finale est également 0. Non seulement, le calcul de f termine (par hypothèse), mais notre gestion du *pipe-line* nous permet de savoir à quel moment exactement ce calcul est terminé.

Preuve de (iii)

Pour un x quelconque de sorte t, nous pouvons représenter par un arbre fini les appels de $\bar{h}$ durant le calcul de $f(x)$. Cela revient à considérer $f(x)$ comme un terme partiellement développé sous forme d'arbre, sans évaluation des occurrences de h. Si, dans cet arbre, nous remplaçons chaque occurrence de h par $\hbar$, et chaque sous-terme y_0, de sorte t_0, qui est une feuille de l'arbre, par $canonic(y_0)$, cette transformation est sémantiquement correcte. Soit δ la profondeur du réseau, c'est-à-dire le nombre de niveaux d'appels récursifs doubles que nous pouvons simuler par la parallélisation "en largeur". Si la profondeur du terme est inférieure ou égale à δ, ce terme représente l'histoire des appels de $\hbar$ durant l'évaluation de $canonic(f(x))$ par les processus F, G, H. Dans ce cas, la transformation est correcte. Sinon, nous allons transformer ce terme en une famille de termes de profondeur inférieure ou égale à δ. Pour ce faire, nous considérons les sous-termes à la profondeur δ. Si un tel sous-terme est *nil*, il demeure inchangé. Si la racine du sous-terme est $\hbar$, nous l'ajoutons à la famille de termes et nous le remplaçons par *nil* à l'occurrence dans le terme original. Nous répétons cette transformation jusqu'à ce que la profondeur de chaque arbre soit inférieure ou égale à δ.



Ce que l'on peut formaliser par une fonction $\mathit{flatten}_\delta$, comme suit :

$$\mathit{flatten}_\delta(t) = \begin{cases} \{t\} & (\mathit{depth}(t) \leqslant \delta) \\ \{t[\tau \leftarrow \mathit{nil}]_{\tau \in occ_\delta(t)}\} \cup (\displaystyle\bigcup_{\tau \in occ_\delta(t)} \mathit{flatten}_\delta(t/\tau)) & (\mathit{depth}(t) > \delta) \end{cases}$$

avec :

$$occ_\delta(t) = \{\tau \mid \tau \in occ(t) \wedge \lg(\tau) = \delta\}$$

Un arbre quelconque de la famille représente l'histoire des appels de $\hbar$ durant un calcul dont l'entrée est fournie à F_1 et l'évaluation de cet arbre considéré comme un terme est la sortie retournée par H_1.

Le processus $PILOT$ compose les résultats de telles évaluations, deux à deux, au moyen de $\hbar$, c'est-à-dire qu'il rétablit les appels de $\hbar$ qui avaient été enlevés lors de la transformation. h est associative et commutative par hypothèse, $\hbar$ est associative et commutative par construction (cf. proposition 5.3). Par conséquent, l'ordre de composition est indifférent. Notre transformation est correcte. □

5.5 Discussion — Améliorations — Variantes

5.5.1 Discussion

Le réseau construit offre deux niveaux de parallélisme : le parallélisme en largeur et le *pipe-line*. Ils résultent d'une vision "maximaliste" de la parallélisation qui est de vouloir garder une décomposition semblable de bout en bout, même lorsqu'elle excède la profondeur du réseau. Une solution serait de ne profiter que du parallélisme en largeur. Alors, les processus $\{G_i\}$ évalueraient fonctionnellement l'image par f de l'entrée. Cette solution est plus simple à mettre en œuvre et ne nécessite pas de pilote, néanmoins, nous lui trouvons deux défauts :

1. Pour des valeurs ayant besoin d'une grande pile de récursivité, les processus $\{G_i\}$ ont à évaluer des expressions beaucoup plus complexes[73] que la décomposition qui est demandée aux processus $\{F_i\}$.

[73] Étant donné que nous n'avons pas effectué d'étude de complexité, nous nous contenterons ici uniquement du sens intuitif de ce mot.

2. De même, il n'est pas évident que les diverses tâches effectuées par les processus $\{G_i\}$ soient bien réparties. Rien n'empêche, par exemple, le processus G_i de devoir effectuer une évaluation plus complexe que celle du processus $G_(i+1)$.

Une autre critique, beaucoup plus importante, est que des sous-termes des données ayant une taille importante peuvent circuler plusieurs fois de H_1 vers le pilote, puis vers F_1, c'est-à-dire être "promenées" plusieurs fois en mouvement tournant. Si cet inconvénient n'est pas trop grave pour des données "simples" (des entiers, par exemple), par contre, il le devient pour des données plus "structurées". Soit la fonction suivante, définie sur des séquences :

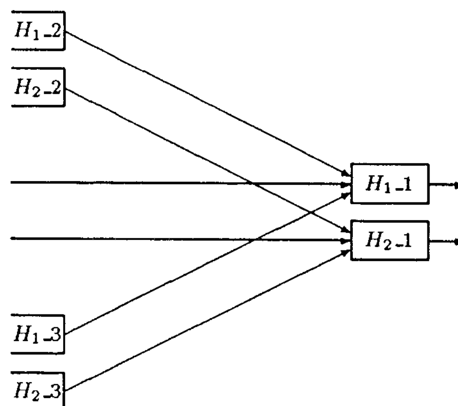
$$f(s) \implies \textbf{if } nil?(s) \textbf{ then } \ldots \textbf{ else } h(f(tail(s)), f(tail(tail(s)))) \textbf{ endif}$$

Soit le terme $f([x_1, \ldots, x_{20}])$ et un réseau de profondeur 3 calculant f. Le terme $[x_1, \ldots, x_{20}]$ se présentera à F_1, suivi des termes $[x_5, \ldots, x_{20}]$ et $[x_6, \ldots, x_{20}]$, puis $[x_9, \ldots, x_{20}]$ et $[x_{10}, \ldots, x_{20}]\ldots$

Il est certain qu'une telle parallélisation ne peut apporter un gain que pour des fonctions très complexes. À ce titre, l'exemple de la fonction *fib* est peut-être "pédagogique", mais certainement pas réaliste. Le principe qui a guidé notre transformation, c'est qu'il n'existe pas moins de raisons de paralléliser les évaluations demandées aux processus $\{G_i\}$ que de raisons de paralléliser celle qui est demandée au processus F_1. Lorsqu'on connaît une mesure de complexité de la fonction que l'on cherche à paralléliser, un compromis qui nous semble acceptable est de tester l'argument d'entrée : si le calcul de son image est d'une complexité que nous considérerons comme suffisamment importante, on applique la technique de parallélisation, sinon, on opte pour une simple évaluation fonctionnelle. Cette mise en œuvre éviterait que des processus soient utilisés pour décomposer et recomposer des calculs simples.

5.5.2 Parallélisation à l'intérieur de *H*

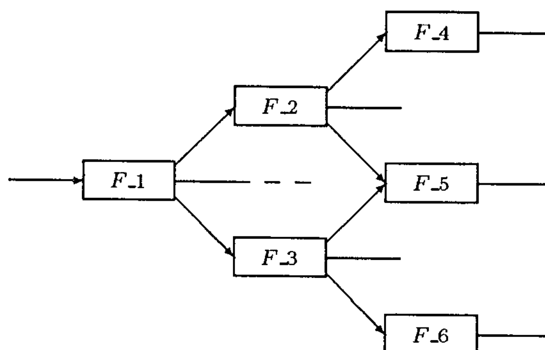
À l'intérieur d'un processus H, sont effectuées deux opérations bien différentes : composition des sous-résultats par $\hbar$, et concaténation des séquences à renvoyer au processus F_1. Il est possible d'effectuer ces opérations en parallèle. L'arbre des processus $\{H_i\}$ devient alors :



Le pilote a alors besoin de gérer deux compteurs : l'un pour les sous-résultats de sorte $ZeroOrOne[t_0]$, l'autre pour les séquences à renvoyer à F_1.

5.5.3 Abaissement du nombre des processus

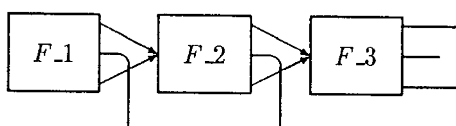
Dans les réseaux décrits précédemment, le nombre de processus croît exponentiellement avec la profondeur. Il est possible de diminuer le nombre de processus nécessaires en construisant un réseau quadratique :



Étant donné que certains processus peuvent recevoir des messages provenant de plusieurs processus, il est alors nécessaire d'éviter les conflits en modifiant les règles. Ainsi :

$$A(x, false) : R(g_1(x))\ S(g_2(x)) ==> INIT \text{ devient : } \left\{ \begin{array}{rcl} A(x, false) & : & R(g_1(x)) ==> B'(x) \\ B'(x) & : & S(g_2(x)) ==> INIT \\ A(x, false) & : & S(g_2(x)) ==> B''(x) \\ B''(x) & : & R(g_1(x)) ==> INIT \end{array} \right.$$

En continuant dans cette voie, le cas extrême d'abaissement du nombre de processus par rapport à la profondeur du réseau est une chaîne linéaire dans laquelle ne subsiste qu'un seul niveau de parallélisme : le *pipe-line*.



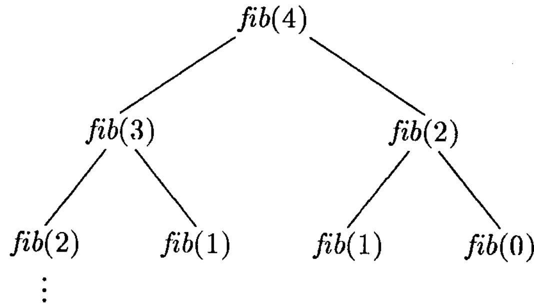
5.6 Élimination des évaluations redondantes

5.6.1 L'idée de base

Nous allons à présent étudier une autre amélioration, que nous traitons à part des précédentes parce qu'elle ne s'applique qu'à un cas particulier de l'équation (5.1) :

$$f(x) \quad ==> \quad \textbf{if } c(x) \textbf{ then } h_0(x) \textbf{ else } h(f(g(x)), f(g(g(x)))) \; \textbf{endif} \tag{5.5}$$

Un point bien connu en programmation fonctionnelle est que certains appels de f sont évalués plusieurs fois. En effet, pour $fib(4)$:



fib(2), *fib*(1), et *fib*(0) sont respectivement évalués 2, 3, et 2 fois. Des techniques de *pliage* et de *dépliage* permettent de résoudre ce problème dans le cadre des transformations de programmes fonctionnels[74]. Nous renvoyons le lecteur intéressé à [32] pour plus de détails et rappelons uniquement la spécification obtenue grâce à ces techniques, généralisée au cas de l'équation (5.5) :

$$
\begin{array}{lllll}
f(x) & ==> & \uparrow 1(f'(x)) & & \\
f'(x) & ==> & \textbf{if}\ c(x) & \textbf{then} & \textbf{let}\ \ u :: h_0(x) \\
 & & & & \qquad \textbf{in}\ \langle u, u\rangle \\
 & & & & \textbf{endlet} \\
 & & & \textbf{else} & \textbf{let}\ \ \langle v, w\rangle :: f'(g(x)) \\
 & & & & \qquad \textbf{in}\ \langle h(v, w), v\rangle \\
 & & & & \textbf{endlet} \\
 & & \textbf{endif} & &
\end{array}
$$

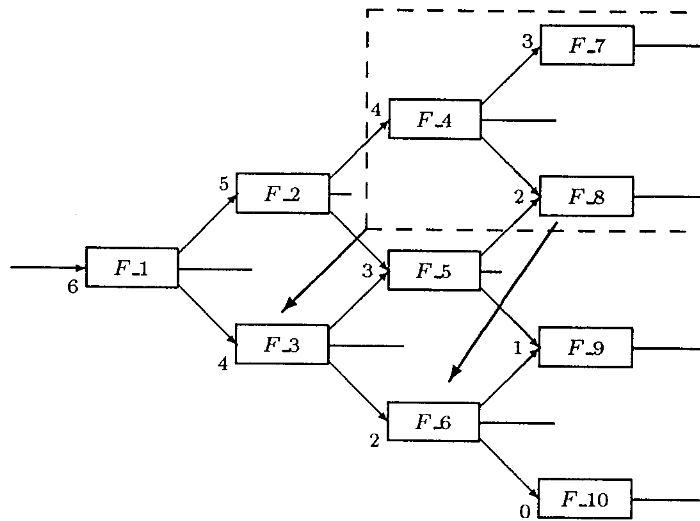
Le profil de f étant $t \rightarrow t_0$, f' est un opérateur auxiliaire de profil $t \rightarrow CP^2[t_0, t_0]$.

La transformation que nous allons développer part du même principe qui est d'éliminer les évaluations redondantes à l'intérieur du réseau quadratique. Si nous partons de ce réseau, c'est parce qu'un certain "regroupement" des mêmes évaluations y existe déjà. Examinons, par exemple, les diverses entrées lors de l'évaluation en parallèle de *fib*(6) :

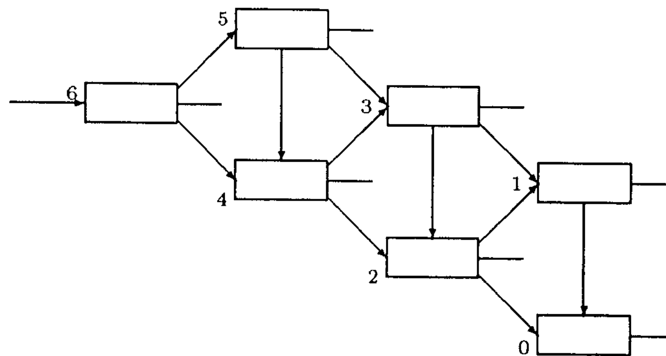
[74]Nous signalons par ailleurs une solution directe à ce problème dans le cadre de la programmation fonctionnelle : celle d'OBJ3, qui offre la possibilité de *mémoriser* les images de certains opérateurs, et, par conséquent, de supprimer pour ces opérateurs les évaluations redondantes. Ainsi, si l'on spécifie *fib* par :

```
obj FIBOLIN is
   protecting NAT .
   op  fib : Nat -> Nat [memo] .
   var i   : Nat .
   eq  fib(i) = if i < 2 then 1 else fib(i - 1) + fib(i - 2)
                fi .
jbo
```

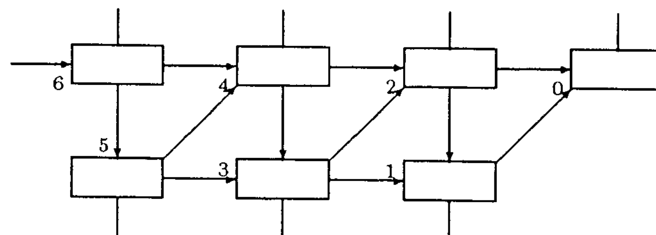
le calcul est linéaire.



Il apparaît que pendant la simulation de la descente récursive, tout processus qui accepte des entrées provenant de deux processus différents effectue en fait le *même* calcul (ainsi, 3 est envoyé à F_5 par F_2 et F_3, 1 est envoyé à F_9 par F_5 et F_6). Nous allons *superposer* les processus qui acceptent des entrées identiques : dans le schéma précédent, nous remplaçons la connexion de F_2 à F_4 par une connexion de F_2 à F_3 et la connexion de F_5 entre F_8 par une connexion entre F_5 et F_6. Après poursuite de cette opération de proche en proche et élimination des processus devenus inaccessibles, nous obtenons le réseau suivant :

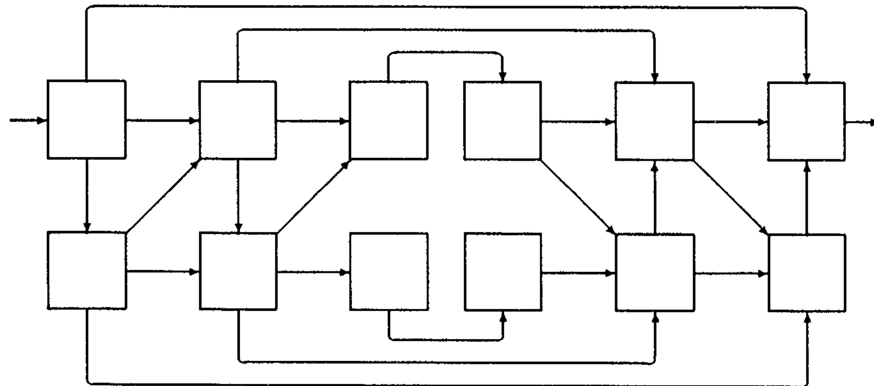


que par commodité, nous représenterons désormais sous une apparence plus "linéaire" :



Dès lors que les processus qui acceptent les mêmes entrées ont été fusionnés, il est à présent possible qu'ils n'effectuent que les calculs indispensables. Le "miroir" de ce réseau s'obtient de la même manière que pour la transformation précédente, c'est-à-dire à partir du "vrai miroir", mais en inversant le sens

de toutes les flèches. Nous obtenons ainsi une double chaîne de processus, dont chaque processus reçoit deux entrées, effectue leur composition par h, et renvoie cette dernière via deux autres connecteurs. Le réseau complet obtenu est :

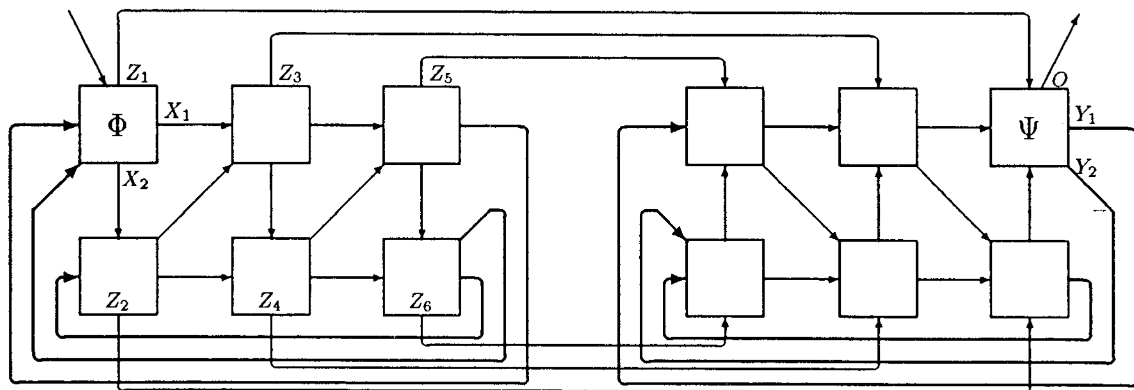


(5.6)

En fait, pour tout $i \geqslant 1$, et pour tout x, lorsque l'évaluation de $f(g^i(x))$ est effectuée, c'est parce qu'elle est demandée et utilisée durant les calculs de $f(g^{i-1}(x))$ et $f(g^{i-2}(x))$. La double entrée pour les processus qui simulent la descente récursive figure la double demande et la double sortie des processus du réseau "miroir" représente la notion duale de double utilisation.

5.6.2 Réseau en couronne

Bien sûr, le même problème des termes trop profonds, dû à l'impossibilité de création dynamique, se pose. Le réseau figuré en (5.6) ne peut évaluer *fib*(x) que pour $x \leqslant 5$. Il est possible d'adopter un schéma de "*pipe-line*", sur le modèle de celui décrit au §5.3.3, mais dans le cas présent, cette solution n'est pas très astucieuse, car elle risque de compromettre le propos de ce réseau qui est de n'évaluer qu'une seule fois chaque appel de f. La solution que nous adoptons consiste à organiser chaque moitié du réseau en couronne. Le réseau devient donc :

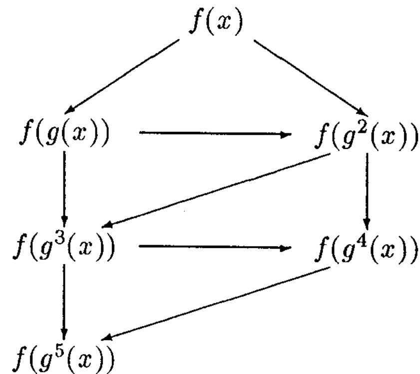


Nous faisons circuler les données tout autour de la couronne "de gauche" et comptons le nombre de messages qui sont envoyés par Φ via les connecteurs X_1 et X_2, avant qu'une réponse soit envoyée au sous-réseau "de droite" via l'un des connecteurs Z_i. Lors du passage par le processus Ψ, exactement autant de messages devront être simultanément envoyés par les connecteurs Y_1 et Y_2 avant que la "réponse finale" soit rendue via O. Si l'on cherche à éviter l'utilisation d'un contrôleur commun, il est nécessaire que le processus Φ envoie un signal au processus Ψ à chaque fois que de nouveaux messages sont envoyés via X_1 et X_2.

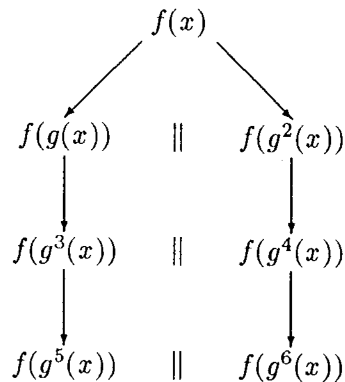
5.6.3 Optimisations possibles

5.6.3.1 Minimisation des échanges de messages

Examinons l'arbre d'appels suivant, inspiré de notre disposition :

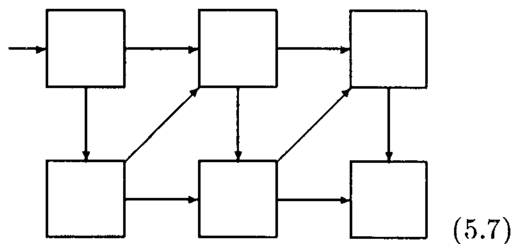


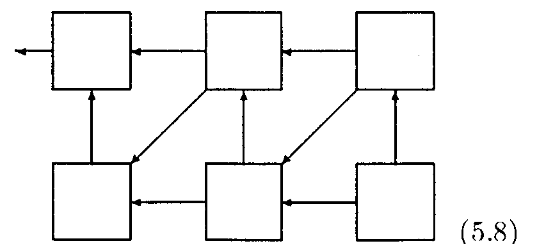
Lorsqu'on simule l'exploration des descentes récursives, $f(x)$ doit au départ demander les deux évaluations de $f(g(x))$ et $f(g^2(x))$. Par contre, il est ensuite inutile que $f(g(x))$ et $f(g^2(x))$ demandent tous deux l'évaluation de $f(g^3(x))$. Il suffit, jusqu'à rencontre de l'arrêt, que $f(g^i(x))$ et $f(g^{i+1}(x))$ demandent respectivement l'évaluation de $f(g^{i+2}(x))$ et $f(g^{i+3}(x))$. Si les deux premières simulations d'appels récursifs débutent simultanément, le maximum de parallélisme est modélisé par le graphe suivant :



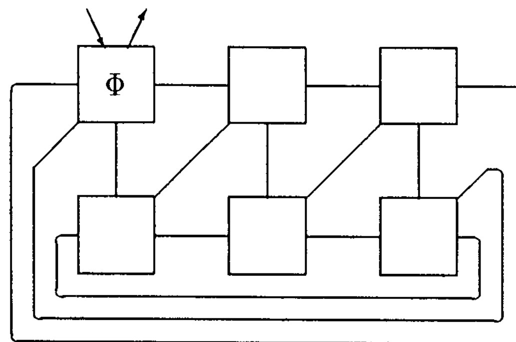
5.6.3.2 Utilisation de la symétrie de la connexion

Si f possède un profil de la forme $t \rightarrow t$, alors les valeurs échangées par les connecteurs sont toutes de sorte t. Si nous nous souvenons qu'il n'existe pas de distinction entre connecteurs d'entrée et connecteurs de sortie, mais que le sens de chaque communication lui est propre, alors nous pouvons utiliser la même moitié de réseau pour tout le calcul. Tout se passe comme si les flèches étaient orientées dans le sens représenté en (5.7) lors de la simulation de la descente récursive, et dans le sens (5.8) pendant les divers calculs de h.


(5.7)


(5.8)

La même remarque peut bien entendu être appliquée au réseau étudié précédemment, mais elle interdit le *pipe-line*. Pour l'application à des termes d'une profondeur quelconque, le même principe de réseau en couronne peut être retenu, avec un compteur présent dans Φ, qui est d'abord incrémenté, puis décrémenté. Le réseau final est alors :



Nous ne donnerons pas ici de preuve de correction de ces derniers réseaux. Signalons toutefois qu'il nous semble possible de relâcher l'hypothèse d'associativité et de commutativité de h.

D'une façon générale, beaucoup de réseaux construits dans ce chapitre étant fortement synchrones, une suite possible de ce travail, que nous n'avons pas traitée, serait d'étudier l'adaptabilité de certaines preuves de réseaux systoliques aux réseaux obtenus au moyen des transformations que nous avons développées.

Un autre point important est que, si, durant ce chapitre, nous avons utilisé la plupart des fonctionnalités de la spécification de processus parallèles communicants en FP2 (utilisation de règles de transition, généricité, symétrie de la connexion, ...), nous n'avons pas abordé le problème de la mise en œuvre des réseaux précédents sur une architecture parallèle réelle. Nous mentionnons ce problème — que nous n'avons pas approfondi — pour faire ressortir qu'un nombre trop élevé de processus est de toute façon irréaliste si l'on cherche à utiliser un processeur pour simuler un seul processus, et que ceci constitue une raison supplémentaire de savoir réaliser la parallélisation d'une fonction avec un nombre relativement restreint de processus. Ainsi, on peut espérer que la topologie du réseau soit assez proche d'une architecture réelle.

Conclusion

Nous voici au terme des chapitres de la thèse. Tout d'abord, nous rappelons quels étaient notre démarche, nos objectifs, et quels sont nos apports.

Notre propos principal était, nous l'avons vu dès l'introduction, d'exposer les principes de la partie fonctionnelle du langage FP2. Du point de vue de l'utilisateur de ce langage, les caractéristiques de cette partie fonctionnelle se distinguent peu de celles d'autres langages de spécification algébrique, en particulier peu de celles de LPG, dont l'expérience et les apports ont largement influencé FP2. Un point spécifique est la mise en œuvre des spécifications exécutables : elle est caractérisée par une approche de compilation qui intègre entièrement les opérateurs génériques. De façon générale, un trait caractéristique important de la mise en œuvre de la partie fonctionnelle de FP2 est que la généricité n'est pas traitée comme un ajout au langage : toutes les sortes, tous les opérateurs sont considérés comme génériques, le cas non paramétré étant compris comme un cas particulier plus simple. Quant à la définition proprement dite du langage, une nouveauté importante est la formulation précise de notions qui auparavant n'étaient introduits qu'opérationnellement (exceptions) ou informellement (instanciations implicites et explicites, modèles génériques). Dans cette optique, les notions de généricité et d'instanciation sont également données de manière plus générale, y compris l'introduction de l'instanciation partielle, qui n'existe pas en LPG. Comme on aura pu le remarquer, aussi bien à propos des exceptions qu'au sujet de la généricité, nous sommes partis d'un "cahier des charges" intuitif, inspiré pour une large part de l'expérience de LPG, et nous avons redéfini, généralisé ces notions pour FP2.

Nous récapitulons à présent nos principaux résultats :

- description des bases de la généricité dans la partie fonctionnelle de FP2, sémantique des types génériques et des modèles génériques, règles de l'instanciation (chapitre 1) ;
- règles de compilation des opérateurs génériques, avec preuve de correction par rapport à la sémantique opérationnelle (chapitre 2) ;
- spécification du traitement des raccourcis de notation (chapitre 3) ;
- théorème de terminaison pour un système de réécriture complété avec les règles qui représentent la propagation des termes exceptionnels (chapitre 4), résultat s'accompagnant d'une technique de linéarisation des systèmes de réécriture (annexe C) ;
- description opérationnelle précise des exceptions et sémantique algébrique ramenant l'étude d'une présentation avec exceptions à celle d'une présentation avec sortes ordonnées (chapitre 4) ;
- exposé de méthodes de transformation de fonctions récursives en processus parallèles (chapitre 5).

En l'état actuel de ce dernier point, qui sort du cadre "purement fonctionnel", il est certain que beaucoup de travail reste à accomplir dans cette voie, avant que la transformation de fonctions en processus parallèles puisse déboucher sur une réalisation effective. Cependant, outre que la donnée des hypothèses de nos transformations tire parti de la généricité suivant un principe très proche de l'utilisation de cette dernière pour décrire des transformations de programmes fonctionnels [9], nous pensons avoir mis en évidence que dans le cas de la simulation de fonctions à double appel récursif, l'utilisation de réseaux dont la topologie est fixée statiquement ne constitue pas une grosse limitation, même si l'on adopte une vision "maximaliste" de la parallélisation.

Comme nous l'avons fait remarquer dans l'introduction et au chapitre 0, les deux parties, programmation fonctionnelle et programmation parallèle, sont définies à travers la même approche : termes et réécriture de termes, et c'est ce qui constitue l'originalité de FP2. La transformation de programmes ou, pour rester dans un cadre plus algébrique, la définition d'équivalences entre fonctions et processus est facilitée par l'homogénéité du langage. Pour bien fixer les bases sémantiques de ces équivalences, une suite de ce travail serait bien évidemment l'étude complète des processus génériques. En approfondissant cette liaison entre fonctions et processus parallèles, on peut espérer relier cette application à certaines études déjà amorcées, d'une part à propos des transformations de programmes fonctionnels [32, 9], d'autre part au sujet de la comparaison et des équivalences entre processus dans le cadre de FP2 [154]. Et, bien sûr, une autre suite opportune à ce travail, orientée vers l'atelier de transformations, serait l'étude de la complexité des programmes parallèles en FP2, qui déboucherait tout naturellement sur l'étude du gain en complexité des transformations dans le cadre de FP2.

Moyennant une définition précise et complète de la sémantique, un autre apanage de l'approche homogène de FP2 est la possibilité d'envisager des démonstrations de théorèmes qui nécessitent de manipuler à la fois des objets de la partie fonctionnelle et ceux de la partie parallèle. C'est un avantage que, par exemple, n'offre pas LOTOS [181], où le formalisme des processus communicants est complètement indépendant de celui des types[75].

En ce qui concerne notre définition des exceptions, définition qui les inscrit dans le cadre des algèbres avec sortes ordonnées, son grand avantage est de permettre l'utilisation de beaucoup de travaux auxquels les sortes ordonnées ont donné lieu. De plus, s'il apparaît que l'introduction des sortes ordonnées ne bouleverse pas le langage, alors nous pensons pouvoir affirmer que notre définition des exceptions constitue un prolongement naturel du formalisme de base. Par contre, si les exceptions permettent d'exprimer le blocage d'un processus particulier dans un réseau, comme nous l'avons vu à la fin du chapitre 4, il nous semble difficile, voire impossible d'être plus ambitieux dans cette voie, de permettre la transmission de valeurs exceptionnelles entre processus sans modifier profondément les bases du langage.

FP2 est avant tout un langage de spécification, mais il a été également destiné à quelques expériences de programmation [104, 157, 8, 107]. C'est dans cette optique que nous comprenons la description du langage comme une décomposition en deux parties : les *facilités* de spécification, et le *noyau* dont on fournit une sémantique complète. C'est le cas de la généricité, où les facilités consistent en les instanciations implicites et explicites. C'est également le cas du traitement de la surcharge, qui rend facile l'utilisation de cette dernière. Dans une certaine mesure, c'est encore le cas pour les exceptions, une présentation avec exceptions étant plus accessible à l'utilisateur, plus directement proche de ce qu'il cherche à modéliser, et surtout plus légère que la présentation avec sortes ordonnées qui en constitue la définition "solide".

[75] La partie fonctionnelle de LOTOS est en fait le langage ACT ONE [59].

On pourra remarquer la multiplicité des notions qui ont été introduites en FP2. Il est vrai qu'elles rendent le langage riche de nombreuses possibilités, il n'est pas moins vrai qu'elles peuvent le rendre long à assimiler et difficile à maîtriser. Nous pensons malgré tout qu'il est possible de l'approfondir par couches successives, les applications simples à spécifier ne nécessitant pas de connaître toutes les possibilités du langage.

Concernant plus directement le prototype réalisé, des autres suites possibles, en vue d'obtenir un véritable atelier de spécification, seraient d'adapter à FP2 des primitives de gestion de définitions, de composition de spécifications, ainsi que des outils d'aide à la mise au point, tant pour la partie fonctionnelle que pour la partie parallèle.

À l'énoncé de toutes ces perspectives, nous concluons en disant que FP2 nous semble au carrefour de beaucoup de directions de recherche actuelles.

Annexes

Annexe A

Rappels de théorie des catégories

Les paradoxes classiques (inexistence d'un ensemble de tous les ensembles, impossibilité de définir un ensemble par une proposition quelconque[76]) ont motivé l'introduction d'objets plus grands que les ensembles, afin de pouvoir considérer la fonctionnalité en se plaçant dans un cadre universel. Historiquement, les catégories ont été introduites pour étudier la topologie de façon algébrique. Toutefois, la formalisation des notions de cette théorie a permis de dégager des applications très variées et la généralité de ses constructions permet la reconstruction, à partir de ce cadre général, de beaucoup d'autres théories, parmi lesquelles la théorie des ensembles, la théorie de la démonstration, la calculabilité, ... En particulier, certaines propriétés des modèles d'une signature, d'une présentation peuvent être étudiées dans ce cadre [76, 77, 78]. Afin de ne pas trop ralentir l'exposé des chapitres, nous avons regroupé dans cette annexe les principales définitions, ainsi que les principaux résultats de théorie des catégories, qui sont utilisés dans l'ouvrage. Ces définitions et résultats sont extraits de [65, 134, 99], et illustrés par quelques exemples "parlants", la quasi-totalité issus de la théorie des ensembles.

A.1 Catégories

A.1.1 Définitions de base

Définition A.1 *Une* **catégorie** $\mathcal{C}$ *est la donnée d'***objets** *(*Ob $\mathcal{C}$*) et de* **flèches**[77] *(*Arr $\mathcal{C}$*).*

Une flèche est caractérisée par sa **source** *et son* **but**, *qui sont tous deux des objets de* $\mathcal{C}$. *On note* $f : A \to B$ *(resp.* $\mathcal{C}(A, B)$*) une flèche* f *(resp. toutes les flèches) de source* A *et de but* B.

On impose de plus que pour tout objet A, *il existe une flèche* $id_A : A \to A$, *et que pour toutes flèches* $f : A \to B$ *et* $g : B \to C$, *il existe une flèche* $g \circ f : A \to C$ *(*$g \circ f$ *est la* **composition** *de* f *et* g*), toutes ces flèches étant assujetties aux axiomes suivants*[78] *:*

$$(\forall f \in \mathcal{C}(A,B)), \quad \left.\begin{array}{c} f \circ id_A \\ id_B \circ f \end{array}\right\} = f$$

$$(\forall f \in \mathcal{C}(A,B)), (\forall g \in \mathcal{C}(B,C)), (\forall h \in \mathcal{C}(C,D)), \quad h \circ (g \circ f) = (h \circ g) \circ f$$

[76] Cf. l'exemple classique : $x \notin x$.

[77] En anglais : *arrows*.

[78] Par abus de langage, nous écrivons $A \in$ Ob $\mathcal{C}$, $f \in$ Arr $\mathcal{C}$, $f \in \mathcal{C}(A, B)$, bien que Ob $\mathcal{C}$, Arr $\mathcal{C}$, $\mathcal{C}(A, B)$ ne soient pas nécessairement des ensembles.

Exemple A.2 *Soit E un ensemble muni d'un préordre*[79] *"$\preceq$". Tout élément de E est un objet de la catégorie $\mathbf{Pre}_{(E,\preceq)}$, et tout couple (e_1, e_2) du graphe de "$\preceq$" forme une flèche $e_1 \to e_2$. La composition de deux flèches $e_1 \to e_2$ et $e_2 \to e_3$ s'obtient par composition des deux couples sous-jacents :*

$$(e_2, e_3) \circ (e_1, e_2) \stackrel{\text{déf}}{=} (e_1, e_3)$$

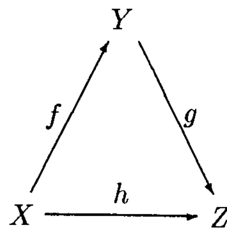
la propriété de transitivité garantissant l'appartenance de (e_1, e_3) au graphe de "$\preceq$".

Exemple A.3 *Soit $\mathcal{U}$ un univers*[80]*, il détermine une catégorie $\mathbf{Ens}_{\mathcal{U}}$ qui admet pour objets les petits ensembles éléments de $\mathcal{U}$ et pour flèches les applications entre ces ensembles.*

Laissant $\mathcal{U}$ implicite, nous noterons désormais $\mathbf{Ens}_{\mathcal{U}}$ par **Set**, *que nous appellerons* **catégorie des petits ensembles.**

Note Les flèches d'une catégorie sont parfois appelées **morphismes**, pour exprimer qu'elles préservent une structure commune des objets.

En théorie des catégories, lorsqu'on désire exprimer des équations entre flèches et compositions de flèches, on préfère, plutôt que de les exhiber directement, les exprimer sous forme d'un diagramme, c'est-à-dire d'un graphe orienté dont les sommets sont étiquetés par des objets et les arcs par des flèches. Un diagramme **commutatif** exprime l'égalité sur les paires de chemins du graphe ayant même origine et même extrémité, en convenant qu'un au moins des chemins a une longueur supérieure à 1. Ainsi, le diagramme commutatif suivant :



exprime que $g \circ f = h$.

[79] Un **préordre** est une relation binaire réflexive et transitive.

[80] La notion d'univers [134] a été introduite afin de pouvoir considérer une large collection d'ensembles tout en évitant les paradoxes classiques. Un **univers** $\mathcal{U}$ est un ensemble satisfaisant les propriétés suivantes de fermeture :

(i) $x \in u \in \mathcal{U} \Longrightarrow x \in \mathcal{U}$,

(ii) $u \in \mathcal{U} \wedge v \in \mathcal{U} \Longrightarrow \begin{cases} \{u, v\} & \in \mathcal{U} \\ (u, v) & \in \mathcal{U} \\ u \times v & \in \mathcal{U} \end{cases}$

(iii) $u \in \mathcal{U} \Longrightarrow \begin{cases} \mathfrak{P}(u) & \in \mathcal{U} \\ \bigcup_{x \in u} x & \in \mathcal{U} \end{cases}$

(iv) $\omega \in \mathcal{U}$

(v) pour $x \in \mathcal{U}$ et $u \subseteq \mathcal{U}$, si $f : x \to u$ est surjective, alors $u \in \mathcal{U}$

où :

- $\mathfrak{P}(u)$ désigne l'ensemble des parties de u,
- $\omega = \{0, 1, 2, \ldots\}$ désigne l'ensemble des ordinaux finis.

Les éléments de $\mathcal{U}$ sont appelés **petits ensembles**.

Un sous-ensemble quelconque d'un univers forme une **classe**. À noter qu'un petit ensemble est une classe, mais que la réciproque est fausse (par exemple, $\mathcal{U}$ n'est pas un petit ensemble). Les classes et les applications entre les classes forment les objets et les flèches d'une catégorie notée **Cls**.

A.1.2 Définitions diverses

Définition A.4 *Une flèche $f : A \to B$ est un* **isomorphisme** *si et seulement si f admet un inverse, c'est-à-dire s'il existe une flèche $f' : B \to A$ telle que $f' \circ f = id_A$ et $f \circ f' = id_B$. On note :*

$$f : A \cong B$$

Si une telle flèche f' existe, elle est unique et notée f^{-1}.

Définition A.5 *Un objet A d'une catégorie C est dit* **initial** *si et seulement si pour tout objet X de C, il existe une flèche unique $A \to X$.*

Exemple A.6 *Dans la catégorie* Set*, $\varnothing$ est un objet initial.*

Proposition A.7 *S'il existe, l'objet initial d'une catégorie est unique à un isomorphisme près.*

Par dualité, on obtient la notion d'objet terminal.

Définition A.8 *Un objet Z d'une catégorie C est dit* **terminal** *si et seulement si pour tout objet X de C, il existe une flèche unique $X \to Z$.*

Exemple A.9 *Dans la catégorie* Set*, tout singleton est un objet terminal.*

Définition A.10 *Soit une flèche $f : A \to B$. On appelle* **rétraction** *de f toute flèche inverse à gauche de f, c'est-à-dire toute flèche $f' : B \to A$ telle que $f' \circ f = id_A$.*

La notion de rétraction s'étend à un cas particulier de la définition précédente, appliqué à deux ensembles reliés par inclusion :

Définition A.11 *Soient A et B deux ensembles tels que $A \subseteq B$, et soit $i : A \hookrightarrow B$, l'injection canonique. Par extension, on appelle* **rétraction** *de B vers A toute rétraction de i.*

A.1.3 Constructions catégoriques

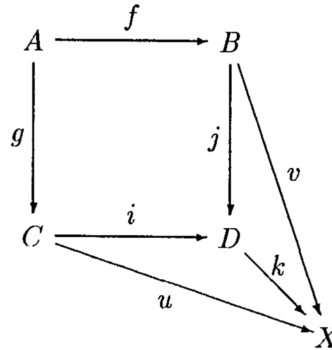
Nous abordons à présent la généralisation des notions ensemblistes de somme minimale et de produit maximal.

A.1.3.1 Somme amalgamée

Définition A.12 *Soit, dans une catégorie quelconque, le diagramme suivant :*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \\ C & & \end{array}$$

La **somme amalgamée**[81] *de f et de g est la donnée d'un objet D et de deux flèches $i : C \to D$ et $j : B \to D$, tels que quels que soient un objet X et des flèches $u : C \to X$ et $v : B \to X$ avec $v \circ f = u \circ g$, il existe une flèche unique $k : D \to X$, telle que le diagramme suivant soit commutatif :*



Si l'objet D existe, il est unique à un isomorphisme près. On note :

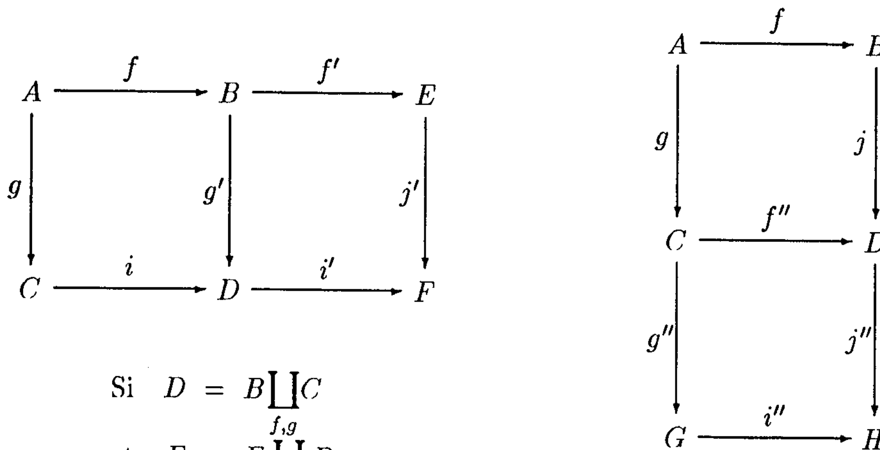
$$D = B \coprod_{f,g} C$$

Exemple A.13 *Dans la catégorie* **Set**, *la somme amalgamée de f et de g est le quotient de la somme disjointe de B et C par la relation $\Re$ définie sur $B \times C$ par $((\forall x \in A), f(x) \Re g(x))$, soit :*

$$B \coprod_{f,g} C = (B \amalg C)/\Re$$

i et j sont alors les injections canoniques.

La composition de deux sommes amalgamées est une somme amalgamée :



$$\begin{aligned} \text{Si } D &= B \coprod_{f,g} C \\ \text{et } F &= E \coprod_{f',g'} D, \\ \text{alors } F &= E \coprod_{f' \circ f, g} C \end{aligned}$$

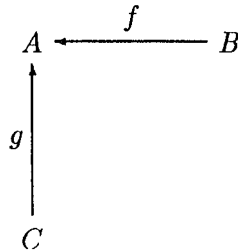
De même :

$$(B \coprod_{f,g} C) \coprod_{f'',g''} G = B \coprod_{f, g'' \circ g} G$$

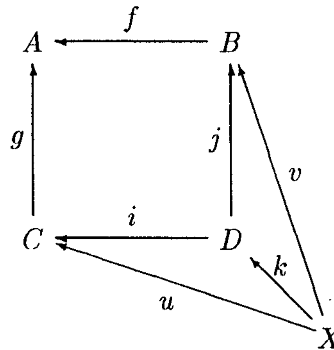
[81] En anglais : *pushout*.

A.1.3.2 Produit fibré

Définition A.14 *Soit, dans une catégorie quelconque, le diagramme suivant :*



Le **produit fibré**[82] *de f et g est la donnée d'un objet D et de deux flèches $i : D \to C$ et $j : D \to B$, tels que quels que soient un objet X et des flèches $u : X \to C$ et $v : X \to B$ avec $f \circ v = g \circ u$, il existe une flèche unique $k : X \to D$, telle que le diagramme suivant soit commutatif :*



Si D existe, il est unique à un isomorphisme près. On note :

$$D = B \prod_{f,g} C$$

Exemple A.15 *Dans la catégorie* Set *:*

$$B \prod_{f,g} C = \{(x,y) \mid x \in B \wedge y \in C \wedge f(x) = g(y)\}$$

i et j sont les projections :

$$i : \begin{array}{ccc} B\prod_{f,g} C & \to & C \\ (x,y) & \mapsto & y \end{array} \qquad j : \begin{array}{ccc} B\prod_{f,g} C & \to & B \\ (x,y) & \mapsto & x \end{array}$$

A.2 Foncteurs — Transformations — Adjonction

A.2.1 Foncteurs

Nous allons introduire les morphismes entre catégories, plus communément appelés foncteurs.

[82]En anglais : *pullback*.

Définition A.16 *Soient $\mathcal{C}_1$ et $\mathcal{C}_2$ deux catégories. Un* **foncteur** $F : \mathcal{C}_1 \to \mathcal{C}_2$ *est la donnée de deux applications*[83] *(également notées F)* Ob $\mathcal{C}_1 \to$ Ob $\mathcal{C}_2$ *et* Arr $\mathcal{C}_1 \to$ Arr $\mathcal{C}_2$ *qui préservent la structure catégorique :*

(i) $(\forall f \in \mathcal{C}_1(A_1, B_1)), F(f) : F(A_1) \to F(B_1)$

(ii) $(\forall A_1 \in \text{Ob } \mathcal{C}_1), F(id_{A_1}) = id_{F(A_1)}$

(iii) $(\forall f \in \mathcal{C}_1(A_1, B_1)), (\forall g \in \mathcal{C}_1(B_1, C_1)), F(g \circ f) = F(g) \circ F(f)$

(On remarquera à propos de (iii) que l'axiome (i) garantit que $F(f)$ et $F(g)$ sont composables dès lors que f et g le sont.)

Pour une catégorie $\mathcal{C}$, le foncteur identité $\mathcal{C} \to \mathcal{C}$ sera noté $id_{\mathcal{C}}$.

Exemple A.17 *Le foncteur puissance* $\mathfrak{P}$: **Set** $\to$ **Set** *associe à un ensemble A l'ensemble* $\mathfrak{P}(A)$ *et à une fonction* $f : A \to B$ *la fonction* $\mathfrak{P}(f) : \mathfrak{A} \to \mathfrak{B}$ *telle que pour tout* $X \subseteq A$, $\mathfrak{P}(f)(X) = f(X)$.

Soient deux foncteurs $F : \mathcal{C}_1 \to \mathcal{C}_2$ et $G : \mathcal{C}_2 \to \mathcal{C}_3$. Il est immédiat de définir $G \circ F : \mathcal{C}_1 \to \mathcal{C}_3$, la **composition** de F et G, par les deux applications $G \circ F$: Ob $\mathcal{C}_1 \to$ Ob $\mathcal{C}_3$ et $G \circ F$: Arr $\mathcal{C}_1 \to$ Arr $\mathcal{C}_3$, et de montrer que cette composition est associative.

Définition A.18 *Soient $\mathcal{C}_1$ et $\mathcal{C}_2$ deux catégories. Un foncteur* $F : \mathcal{C}_1 \to \mathcal{C}_2$ *est un* **isomorphisme de catégories** *s'il existe un foncteur* $F' : \mathcal{C}_2 \to \mathcal{C}_1$ *tel que* $F' \circ F = id_{\mathcal{C}_1}$ *et* $F \circ F' = id_{\mathcal{C}_2}$. *Par analogie avec les flèches, F' (qui est unique) est noté F^{-1}.*

Décrivons à présent certains foncteurs très utilisés pour passer d'une structure à une autre structure “moins riche”. Lorsque deux catégories $\mathcal{C}_1$ et $\mathcal{C}_2$ sont telles que toute structure de $\mathcal{C}_1$ est une structure de $\mathcal{C}_2$, les injections canoniques déterminent un foncteur $\mathcal{C}_1 \to \mathcal{C}_2$, appelé **foncteur d'oubli**.

Exemple A.19 *Considérons les petits ensembles munis d'une loi de composition interne. Soit* **Mon** *(resp.* **Grp***) la catégorie dont les objets sont les petits monoïdes (resp. les petits groupes) et les flèches les morphismes de monoïdes (resp. les morphismes de groupes). Le foncteur d'oubli* **Grp** $\to$ **Mon** *fait correspondre à tout groupe la structure de monoïde sous-jacente en “oubliant” l'opération d'inversion.*

Note La notion de foncteur permet dès lors de définir des *catégories de catégories*, ce qui permet de “réfléchir la théorie dans la méta-théorie”, ainsi que le mentionne Gérard Huet dans [99]. On évite les paradoxes classiques (catégorie de toutes les catégories) en ne considérant que certaines catégories comme objets. Une **petite catégorie** (resp. **grande catégorie**) [134] est une catégorie dont, d'une part, les objets forment un petit ensemble (resp. une classe), d'autre part, les flèches forment un petit ensemble (resp. une classe). On définit alors **Cat** (resp. **Cat'**), la catégorie dont les objets sont les petites catégories (resp. les grandes catégories) et les flèches les foncteurs entre catégories.

[83] Cet emprunt au vocabulaire ensembliste est ici un abus de langage qui revient à traiter Ob $\mathcal{C}_1$, Ob $\mathcal{C}_2$, Arr $\mathcal{C}_1$, Arr $\mathcal{C}_2$ comme des ensembles alors qu'en fait, ils ne le sont pas nécessairement (cf. note 78, p. 177).

A.2.2 Transformations

Définition A.20 *Soient C_1 et C_2 deux catégories. Soient également deux foncteurs $F, G : C_1 \to C_2$.*

- *Une* **transformation** *τ de F vers G, notée $\tau : F \dot{\to} G$, est une application qui fait correspondre à tout objet X_1 de C_1 une flèche $\tau_{X_1} : F(X_1) \to G(X_1)$.*
- *Soit A_1 un objet de C_1. τ est dite* **naturelle** *en A_1 si et seulement si pour toute flèche $f : A_1 \to B_1$, le diagramme suivant est commutatif :*

$$\begin{array}{ccccc} A_1 & \quad & F(A_1) & \xrightarrow{\tau_{A_1}} & G(A_1) \\ \downarrow f & & \downarrow F(f) & & \downarrow G(f) \\ B_1 & & F(B_1) & \xrightarrow{\tau_{B_1}} & G(B_1) \end{array}$$

- *τ est dite naturelle si et seulement si elle est naturelle en tout objet A_1 de C_1.*

Exemple A.21 *Nous plaçant à nouveau dans la catégorie* **Set**, *nous notons A^X l'ensemble de toutes les applications $X \to A$. Soit le foncteur F :* **Set** $\to$ **Set**, *défini comme suit :*

$$\begin{array}{lcl} (\forall A \in \text{Ob } \mathbf{Set}), F(A) & = & A^X \times X \\ (\forall f : A_1 \to A_2), F(f) & : & {A_1}^X \times X \to {A_2}^X \times X \text{ est telle que } F(f)(h_1, x) = (f \circ h_1, x) \end{array}$$

pour $h_1 \in {A_1}^X, x \in X$.

La transformation d'évaluation $eval : F \dot{\to} id_{\mathbf{Set}}$ définie, pour $h \in A^X, x \in X$, par :

$$eval_A(h, x) = h(x)$$

est une transformation naturelle.

Définition A.22 *Une transformation naturelle $\tau : F \dot{\to} G$ est un* **isomorphisme naturel** *si pour tout objet X_1 de C_1, τ_{X_1} est un isomorphisme. On note :*

$$\tau : F \cong G$$

Note La notion de transformation naturelle se rattache à celle de **catégorie de foncteurs**. Soient C_1 et C_2 deux catégories. ${C_2}^{C_1}$ est la catégorie dont les objets sont les foncteurs $C_1 \to C_2$ et les flèches les transformations naturelles entre ces foncteurs.

A.2.3 Construction libre

La notion d'algèbre libre sur une signature, vue dans la proposition 0.8, est un cas particulier d'une notion catégorique : la construction libre.

Définition A.23 *Soient C_1 et C_2 deux catégories. Soient également $U : C_2 \to C_1$ un foncteur et $X_1 \in \text{Ob } C_1$. Un objet $F(X_1)$ de C_2 est une* **construction libre sur X_1 engendrée par** *U s'il existe une flèche $\tau_{X_1} : X_1 \to U(F(X_1))$, telle que pour tout objet A_2 de C_2 et toute flèche $f : X_1 \to U(A_2)$, alors il existe une unique flèche $f' : F(X_1) \to A_2$ telle que le diagramme suivant soit commutatif :*

$$\begin{array}{ccc} X_1 & \xrightarrow{f} & U(A_2) \\ \tau_{X_1} \downarrow & \nearrow U(f') & \\ U(F(X_1)) & & \end{array}$$

Dans le cas de l'algèbre des termes sur une signature $\Sigma = (S, \Omega)$, engendrée par un ensemble de variables V, U est le foncteur d'oubli U_S : $\mathbf{ALG}_\Sigma \rightarrow \mathbf{Set}_S$, qui envoie chaque Σ-algèbre vers la famille indicée par S de ses supports, et τ_{X_1} l'injection canonique $V \hookrightarrow U_S(T_\Sigma(V))$.

Exemple A.24 *Considérons les catégories* **Set** *et* **Mon** *(cf. exemple A.19). Soit U le foncteur d'oubli* **Mon** $\rightarrow$ **Set**, *qui fait correspondre à tout monoïde l'ensemble sous-jacent. Soit X un ensemble (objet de* **Set***), posons $F(X) = X^*$, le monoïde libre des mots formés avec les éléments de X, muni de l'opération de concaténation. $\tau_X : X \rightarrow U(X^*)$ fait correspondre à tout élément x de X le mot (x) de X^*. Considérons $M \in$ Ob* **Mon** *et $f : X \rightarrow U(M)$. Notons λ le mot vide de X^*, "$\bullet$" la loi de monoïde de M, e son élément neutre, et définissons $g : X^* \rightarrow M$ comme suit :*

$$\begin{array}{rcl} g(\lambda) & = & e \\ g((x_1 \ldots x_p)) & = & f(x_1) \bullet \ldots \bullet f(x_p) \end{array}$$

g est l'unique flèche de **Mon** *qui réalise $U(g) \circ \tau_X = f$. X^* est une construction libre sur X engendrée par U.*

Définition A.25 *Soient C_1 et C_2 deux catégories, et soit un foncteur $U : C_2 \rightarrow C_1$. Si pour tout $A \in$ Ob C_1, il existe $F(A)$, construction libre sur A engendrée par U, alors cette construction libre s'étend aux flèches de C_1 : pour toute flèche $h : A \rightarrow B$ ($B \in$ Ob C_1), la flèche $F(h) : F(A) \rightarrow F(B)$ est telle que le diagramme suivant soit commutatif :*

$$\begin{array}{ccc} A & \xrightarrow{h} & B \\ \tau_A \downarrow & & \downarrow \tau_B \\ U \circ F(A) & \xrightarrow{U \circ F(h)} & U \circ F(B) \end{array}$$

$F : C_1 \rightarrow C_2$ est unique à un isomorphisme naturel près, il est appelé **foncteur libre** *engendré par U. $\tau : id_{C_1} \dot{\rightarrow} U \circ F$ est une transformation naturelle appelée alors* **transformation universelle.**

Proposition A.26 *L'image d'un objet initial par un foncteur libre est un objet initial.*

A.2.4 Adjonction

Conceptuellement, on peut considérer que l'adjonction est une notion dérivée de la notion de construction libre. Considérons C_1 et C_2 deux catégories, $F : C_1 \rightarrow C_2$ et $U : C_2 \rightarrow C_1$ deux foncteurs,

$A_1 \in \text{Ob } \mathcal{C}_1$. Nous sommes en présence d'une adjonction dans le cas où quel que soit $A_2 \in \text{Ob } \mathcal{C}_2$, à toute flèche $f : A_1 \to U(A_2)$ correspond une flèche $f' : F(A_1) \to A_2$, telle que $F(A_1)$ soit une construction libre engendrée par U, et, de plus, telle que la fonction φ qui réalise $\varphi(f') = f$ soit bijective. C'est ce que traduit, de façon plus générale, la définition suivante.

Définition A.27 *Soient $\mathcal{C}_1$ et $\mathcal{C}_2$ deux catégories. Une* **adjonction** *de $\mathcal{C}_1$ vers $\mathcal{C}_2$, notée :*

$$\langle F, G, \phi \rangle : \mathcal{C}_1 \rightharpoonup \mathcal{C}_2$$

est la donnée de deux foncteurs F et G tels que :

$$\mathcal{C}_1 \underset{G}{\overset{F}{\rightleftarrows}} \mathcal{C}_2$$

et de ϕ une fonction qui fait correspondre à tout couple d'objets A_1 de $\mathcal{C}_1$ et A_2 de $\mathcal{C}_2$ une bijection $\phi_{A_1,A_2} : \mathcal{C}_2(F(A_1), A_2) \cong \mathcal{C}_1(A_1, G(A_2))$ naturelle en A_1 et en A_2, c'est-à-dire telle que :

$$\begin{array}{ccc} X_1 & & F(X_1) \\ \downarrow f_1 & & \downarrow F(f_1) \\ A_1 & & F(A_1) \\ \downarrow \phi_{A_1,A_2}(h) & . & \downarrow h \\ G(A_2) & & A_2 \\ \downarrow G(f_2) & & \downarrow f_2 \\ G(X_2) & & X_2 \end{array} \qquad \begin{array}{rcl} \phi_{A_1,A_2}(h) \circ f_1 & = & \phi_{X_1,A_2}(h \circ F(f_1)) \\ G(f_2) \circ \phi_{A_1,A_2}(h) & = & \phi_{A_1,X_2}(f_2 \circ h) \end{array}$$

On dit que F est **adjoint à gauche** de G, et que G est **adjoint à droite** de F. On déduit alors de F et de G une transformation naturelle $\tau : id_{\mathcal{C}_1} \dot{\to} G \circ F$, appelée **unité** de l'adjonction.

Dans le cas d'un foncteur libre $F : \mathcal{C}_1 \to \mathcal{C}_2$ engendré par un foncteur $U : \mathcal{C}_2 \to \mathcal{C}_1$, alors $\langle F, U, \phi \rangle : \mathcal{C}_1 \rightharpoonup \mathcal{C}_2$ est une adjonction, avec :

$$(\forall A_1 \in \text{Ob } \mathcal{C}_1), (\forall A_2 \in \text{Ob } \mathcal{C}_2), (\forall f \in \mathcal{C}_2(F(A_1), A_2)), \phi_{A_1,A_2}(f) = U(f) \circ \tau_{A_1}$$

Exemple A.28 *Soient A, B, X trois ensembles. L'isomorphisme bien connu :*

$$B^{A \times X} \cong (B^X)^A$$

est un cas particulier d'adjonction. Considérons les foncteurs F, G : **Set** $\to$ **Set**, *tels que :*

$$\begin{array}{rcl} F(A) & = & A \times X \\ G(B) & = & B^X \end{array}$$

$\langle F, G, \phi \rangle$: **Set** $\rightharpoonup$ **Set** *constitue une adjonction, avec :*

$$\phi_{A,B}(f)(a)(x) = f(a,x)$$

pour $f \in B^{A \times X}, a \in A, x \in X$.

A.2.5 Interprétation catégorique des importations de définitions

Revenant à la spécification algébrique, nous allons maintenant présenter brièvement l'importation de présentations avec le formalisme des catégories. Soient $\mathcal{P} = (S, \Omega, E)$ et $\mathcal{P}' = \mathcal{P} \uplus (S', \Omega', E')$ — rappelons que "$\uplus$" désigne l'union disjointe. Posons en outre $\Sigma = (S, \Omega)$ et $\Sigma' = (S \uplus S', \Omega \uplus \Omega')$. Les relations entre les catégories $\mathbf{ALG}_{\mathcal{P}}$ et $\mathbf{ALG}_{\mathcal{P}'}$ sont données par le **foncteur de synthèse** F et le **foncteur d'oubli** U :

$$\mathbf{ALG}_{\mathcal{P}} \underset{U}{\overset{F}{\rightleftarrows}} \mathbf{ALG}_{\mathcal{P}'}$$

Soit une algèbre $A' \in \mathrm{Ob}\ \mathbf{ALG}_{\mathcal{P}'}$, avec $A' = (A'_{S \uplus S'}, A'_{\Omega \uplus \Omega'})$. Le foncteur d'oubli U la restreint aux supports indicés par S et aux ensembles de fonctions qui correspondent aux déclarations de Ω :

$$U(A') = (A'_S, A'_\Omega)$$

Soient deux algèbres $A', B' \in \mathrm{Ob}\ \mathbf{ALG}_{\mathcal{P}'}$, et soit un Σ'-homomorphisme $h' : A' \to B'$, défini par la famille d'applications $\{h'_s : A'_s \to B'_s \mid s \in S \uplus S'\}$. $U(h') : U(A') \to U(B')$ est défini par restriction de h', par la famille $\{h'_s : A'_s \to B'_s \mid s \in S\}$.

Soit une algèbre $A \in \mathrm{Ob}\ \mathbf{ALG}_{\mathcal{P}}$, le foncteur de synthèse F lui fait correspondre la $\mathcal{P}'$-algèbre minimale[84] parmi les $\mathcal{P}'$-algèbres extensions de A :

$$F(A) = \frac{T_{\Sigma'}(U_S(A))}{\equiv_{E \uplus E', \sim_A}}$$

où :

- U_S est le foncteur d'oubli $\mathbf{ALG}_{\mathcal{P}} \to \mathbf{Set}_S$,
- $T_{\Sigma'}(U_S(A))$ est une algèbre de termes que l'on construit en traitant les éléments de A comme des variables,
- "$\equiv_{E \uplus E', \sim_A}$" est la plus petite congruence qui contient d'une part toutes les instances fermées des équations de E et de E', d'autre part la relation "$\sim_A$", définie par :

  $$t_1 \sim_A t_2 \overset{\text{déf}}{\Longleftrightarrow} (t_1, t_2 \in T_\Sigma(U_S(A))) \wedge eval_A(t_1) = eval_A(t_2)$$

  $eval_A : T_\Sigma(U_S(A)) \to A$ est appelé **Σ-homomorphisme d'évaluation dans** A et est défini comme suit :

  $$\begin{aligned} eval_A(t) &= t & (t \in U_S(A)) \\ eval_A(\omega) &= A_\omega \\ eval_A(\omega(t_1, \ldots, t_n)) &= A_\omega(eval_A(t_1), \ldots, eval_A(t_n)) \end{aligned}$$

[84] Au sens où, pour X et Y deux algèbres, on pose $X \leq Y$ si et seulement s'il existe un homomorphisme $X \to Y$.

$F(A)$ est une construction libre sur $U_S(A)$.

Soient deux algèbres $A, B \in$ Ob $\mathbf{ALG}_{\mathcal{P}}$, et soit un Σ-homomorphisme $h : A \to B$. On déduit de h une assignation $\nu : U_S(A) \to F(B)$. On obtient ensuite $F(h) : F(A) \to F(B)$ par extension de ν en utilisant la propriété de construction libre de $F(A)$.

Note Le foncteur libre sémantique d'une présentation paramétrée (cf. chapitre 1) est construit de façon identique.

Ces foncteurs F et U sont adjoints. L'unité de l'adjonction :

$$\tau : id_{\mathbf{ALG}_{\mathcal{P}}} \dot{\to} U \circ F$$

induit, pour toute algèbre $A \in$ Ob $\mathbf{ALG}_{\mathcal{P}}$, un Σ-homomorphisme $A \to U \circ F(A)$, qui permet d'étudier l'effet sur la $\mathcal{P}$-algèbre A des définitions de (S', Ω', E'). On peut en particulier donner une nouvelle caractérisation de la notion de protection, vue au §0.1.4.1. Ainsi, la sorte s de S est **protégée** dans $\mathcal{P}'$ si et seulement si :

$$U \circ F(T_{\mathcal{P}_s}) = T_{\mathcal{P}_s}$$

$\mathcal{P}$ est protégée dans $\mathcal{P}'$ si $U \circ F(T_{\mathcal{P}}) = T_{\mathcal{P}}$.

Exemple A.29 *Spécifions les ensembles $\mathbb{N}$ et $\mathbb{Z}/2\mathbb{Z}$:*

presentation $\mathcal{P}$ **is**
  sorts $= \{Nat\}$
  opns $0 \quad : \; - \quad \to \; Nat$
     $succ \; : \; Nat \; \to \; Nat$
end $\mathcal{P}$

presentation $\mathcal{P}'$ **is** $\mathcal{P}$ ⊎
  vars $n \; : \; Nat$
  eqns
    <> $succ(succ(n)) \; == \; n$
end $\mathcal{P}'$

$$T_{\mathcal{P}_{Nat}} = \{0, succ(0), succ(succ(0)), \ldots\} \text{ d'où } T_{\mathcal{P}_{Nat}} \cong \mathbb{N}$$

$$F(T_{\mathcal{P}_{Nat}}) \cong \{0, succ(0)\} \text{ d'où } U \circ F(T_{\mathcal{P}_{Nat}}) \cong \mathbb{Z}/2\mathbb{Z}$$

Nat n'est pas protégée dans $\mathcal{P}'$: $U \circ F(T_{\mathcal{P}_{Nat}}) \neq T_{\mathcal{P}_{Nat}}$.

Signalons, pour clore ces rappels, que Gilles Bernot a dégagé dans [11] quelques propriétés "peu souhaitables" du foncteur de synthèse, montrant que l'utilisation de la théorie des catégories en spécification algébrique n'est pas exempte de pièges.

Annexe B

Exemples

Nous allons donner ici une “mini-bibliothèque” d'exemples de spécifications fonctionnelles en FP2. Quelques-unes d'entre elles ont été rencontrées dans les chapitres précédents, parfois sous une forme abrégée.

B.1 Propriétés et types “de base”

La plupart des spécifications suivantes sont fournies dans la bibliothèque initiale du compilateur FP2. Ce dernier assure également, pour certains types (*Nat*, *Seq*, *Char*, *String*), le traitement des formats externes usuels (entiers sous forme décimale, séquences délimitées par “[” et “]”, chaînes de caractères entre doubles guillemets) — cf. §2.1.4. C'est la seule facilité supplémentaire qu'offrent ces types prédéfinis, par rapport aux types que peut spécifier un utilisateur.

Exemple B.1 *Propriété Ftype (*Formal type*) et opérateur d'identité.*

```
prop Ftype[t / ]
endprop

--  Dans l'en-tête d'un module, "[t]" est un
--  raccourci pour "req Ftype[t / ]".
```

```
enr [t]
   opns  identity  :  t  →  t
   vars  x         :  t
   rules
      <>  identity(x)  ==>  x
endenr
```

Rappelons que le traitement de la propriété *Ftype* est entièrement pris en charge par le compilateur FP2 (déclaration d'un modèle à chaque définition d'un nouveau type, relations d'inclusion pour une propriété admettant des sortes formelles dans sa signature — cf. note du §3.1.1).

Exemple B.2 *Type Bool des booléens.*

type *Bool*

cons	$true, false$	:	$-$		
opns	not	:	$Bool$	$\rightarrow$	$Bool$
	$and, or, xor, \Rightarrow, \Leftrightarrow$	:	$Bool \times Bool$	$\rightarrow$	$Bool$
vars	b, b_0	:	$Bool$		

rules

<>	$not\ true$	==>	$false$
<>	$not\ false$	==>	$true$
<>	$true\ and\ b$	==>	b
<>	$false\ and\ b$	==>	$false$
<>	$true\ or\ b$	==>	$true$
<>	$false\ or\ b$	==>	b
<>	$true\ xor\ b$	==>	$not\ b$
<>	$false\ xor\ b$	==>	b
<>	$b \Rightarrow b_0$	==>	$(not\ b)\ or\ b_0$
<>	$b \Leftrightarrow b_0$	==>	$(b \Rightarrow b_0)\ and\ (b_0 \Rightarrow b)$

endtype

Rappelons que tous les opérateurs d'arité 2 peuvent être infixés.

Exemple B.3 *Type Nat des entiers naturels.*

```
type Nat
  cons  zero               : –
        succ               : Nat
  opns  pred               : Nat        →  Nat
        =, <, ⩽, >, ⩾      : Nat × Nat  →  Bool
        +, −, ∗, div, rem  : Nat × Nat  →  Nat
        div-rem            : Nat × Nat  →  CP²[Nat, Nat]  --  Produit cartésien,
        zero?              : Nat        →  Bool           --  cf. exemple B.10.
  vars  i, j, k, l, m      : Nat
  prcds
     <>  i div-rem 0  ==>  !zero-divide  --  Par défaut, la précondition est "true".
  rules
     <>                  pred(0)  ==>  !undefined-value
     <>            pred(succ(i))  ==>  i
     <>        0    =   0         ==>  true
     <>        0    =   succ(j)   ==>  false
     <>  succ(i)    =   0         ==>  false
     <>  succ(i)    =   succ(j)   ==>  i = j
     <>        i    <   0         ==>  false
     <>        0    <   succ(j)   ==>  true
     <>  succ(i)    <   succ(j)   ==>  i < j
     <>        i    ⩽   j         ==>  (i < j) or (i = j)
     <>        i    >   j         ==>  j < i
     <>        i    ⩾   j         ==>  j ⩽ i
     <>        0    +   j         ==>  j
     <>  succ(i)    +   j         ==>  succ(i + j)
     <>        i    −   0         ==>  i
     <>        0    −   succ(j)   ==>  !undefined-value
                                       --  Rappelons que la déclaration des opérateurs
                                       --  d'exception est facultative.
     <>  succ(i)    −   succ(j)   ==>  i − j
     <>        0    ∗   j         ==>  0
     <>  succ(i)    ∗   j         ==>  (i ∗ j) + j
     <>        i   div  j         ==>  ↑1(i div-rem j)
     <>        i   rem  j         ==>  ↑2(i div-rem j)
                                       --  L'opérateur "div-rem" permet de mener de front
                                       --  le calcul du quotient et du reste.
     <>        0  div-rem  j      ==>  ⟨0, 0⟩
     <>  succ(i)  div-rem  j      ==>  let  ⟨k, l⟩  ::  i div-rem j,
                                             m      ::  succ(l)
                                            in      if m = j then ⟨succ(k), 0⟩ else ⟨k, m⟩
                                                    endif
                                       endlet
     <>                zero?(0)   ==>  true
     <>          zero?(succ(i))   ==>  false
endtype
```

Exemple B.4 *Type Seq des séquences linéaires.*

```
type Seq[t]
  cons  nil              :  -
        <+               :  t       ×  Seq[t]
  opns  head             :  Seq[t]                    →  t
        tail, reverse    :  Seq[t]                    →  Seq[t]
        nil?             :  Seq[t]                    →  Bool
        +                :  Seq[t]  ×  Seq[t]         →  Seq[t]
        +>               :  Seq[t]  ×  t              →  Seq[t]
        length           :  Seq[t]                    →  Nat
        sub-seq          :  Seq[t]  ×  Nat  ×  Nat    →  Seq[t]
  priv  extract-sub-seq  :  Seq[t]  ×  Nat            →  Seq[t]   -- extract-sub-seq est un
  vars  x                :  t                                     -- opérateur privé (caché),
        s, s0            :  Seq[t]                                -- c'est-à-dire invisible en
        i, j             :  Nat                                   -- dehors du corps de ce
                                                                  -- module.
  rules
    <>                         head(nil)  ==>  !undefined-value
    <>                    head(x <+ s)  ==>  x
    <>                         tail(nil)  ==>  !undefined-value
    <>                    tail(x <+ s)  ==>  s
    <>                      reverse(nil)  ==>  nil
    <>                 reverse(x <+ s)  ==>  reverse(s) +> x
    <>                          nil?(nil)  ==>  true
    <>                     nil?(x <+ s)  ==>  false
    <>                          nil + s0  ==>  s0
    <>                    (x <+ s) + s0  ==>  x <+ (s + s0)
    <>                            s +> x  ==>  s + [x]
    <>                       length(nil)  ==>  0
    <>                  length(x <+ s)  ==>  succ(length(s))
                                              -- sub-seq(s, i, j) : on extrait j éléments de la
                                              -- séquence s à partir de la position i, les
                                              -- positions étant comptées à partir de 0.
    <>                   sub-seq(s, 0, j)  ==>  extract-sub-seq(s, j)
    <>           sub-seq(nil, succ(i), j)  ==>  !undefined-value
    <>        sub-seq(x <+ s, succ(i), j)  ==>  sub-seq(s, i, j)
    <>               extract-sub-seq(s, 0)  ==>  nil
    <>        extract-sub-seq(nil, succ(i))  ==>  !undefined-value
    <>   extract-sub-seq(x <+ s, succ(i))  ==>  x <+ extract-sub-seq(s, i)
endtype
```

Exemple B.5 *Propriété d'égalité — Égalité des séquences.*

```
prop Equality[t / eq]
  opns  eq       :  t × t  →  Bool
  vars  x, y, z  :  t
  eqns
    <>                                x eq x  ==  true
    <>                                x eq y  ==  y eq x
    <>   ((x eq y) and (y eq z)) ⇒ (x eq z)  ==  true
endprop
```

```
model EQBOOL is Equality[Bool / ⇔]
endmodel
```

```
model EQNAT is Equality[Nat / =]
endmodel
```

```
enr req Equality[t / eq]
   opns   =        :  Seq[t] × Seq[t]   →  Bool
   vars   x1, x2   :  t
          s1, s2   :  Seq[t]
   rules
      <>          nil  =  nil            ==>  true
      <>          nil  =  (x2 <+ s2)     ==>  false
      <>   (x1 <+ s1)  =  nil            ==>  false
      <>   (x1 <+ s1)  =  (x2 <+ s2)     ==>  if x1 eq x2 then s1 = s2 else false
                                              endif
endenr
```

```
model EQSEQ req Equality[t / eq] is Equality[Seq[t] / =]
endmodel
```

Exemple B.6 *Relations d'ordre — Tri rapide (*quicksort*) — Ordre lexicographique sur les séquences.*

```
prop Partial-Order[t / rel, eq]
   opns   rel, eq  :  t × t  →  Bool
   vars   x, y, z  :  t
   eqns
      <>                                  x rel x  ==  true
      <>                (x rel y) and (y rel x)  ==  x eq y
      <>   ((x rel y) and (y rel z)) ⇒ (x rel z)  ==  true
   includes  Equality[t / eq]
endprop
```

```
prop Total-Order[t / rel, eq]
   opns   rel, eq  :  t × t  →  Bool
   vars   x, y     :  t
   eqns
      <>  (x rel y) or (y rel x)  ==  true
   includes  Partial-Order[t / rel, eq]
endprop
```

```
model ORDBOOL is Total-Order[Bool / ⇒, ⇔]
endmodel
```

```
enr Quicksort req Total-Order[t / rel, eq]
   opns  sort   : Seq[t]       →  Seq[t]
   priv  split  : t × Seq[t]   →  CP²[Seq[t], Seq[t]]  -- L'opérateur split est invisible en dehors du
   vars  x, y   : t                                    -- module Quicksort.
         s, s0  : Seq[t]
   rules
      <>          sort(nil)  ==>  nil
      <>       sort(x <+ s)  ==>  let ⟨s, s0⟩ :: split(x, s)
                                  in sort(s) + (x <+ sort(s0))
                                  endlet
      <>       split(x, nil)  ==>  ⟨nil, nil⟩
      <>   split(x, y <+ s)   ==>  let ⟨s, s0⟩ :: split(x, s)
                                   in  if  y rel x then ⟨y <+ s, s0⟩
                                           else ⟨s, y <+ s0⟩
                                       endif
                                   endlet
endenr
```

```
enr req Total-Order[t / rel, eq]
   opns  ⩽       : Seq[t] × Seq[t]  →  Bool
   vars  x1, x2  : t
         s1, s2  : Seq[t]
   rules
      <>         nil  ⩽  s2           ==>  true
      <>  (x1 <+ s1)  ⩽  nil          ==>  false
      <>  (x1 <+ s1)  ⩽  (x2 <+ s2)   ==>  if x1 eq x2 then s1 ⩽ s2 else x1 rel x2
                                            endif
endenr
```

```
model ORDSEQ req Total-Order[t / rel, eq] is Total-Order[Seq[t] / ⩽, =]
endmodel
```

B.2 Exemples divers

B.2.1 Programmation "à la FP"

Exemple B.7 *Opérateurs génériques permettant d'écrire des spécifications "à la FP" [4, 177], c'est-à-dire sans variables.*

```
prop Null-Arity[t1, t2 / c]
   opns  c  :  –  →  t2
endprop
```

```
enr Constant req Null-Arity[t1, t2 / c]
   opns  cst  :  t1  →  t2
   vars  x    :  t1
   rules
      <>  cst(x)  ==>  c
endenr
```

```
prop Two-Operators[t1, t2, t3 / g, f]
   opns  f  :  t1  →  t2
         g  :  t2  →  t3
endprop
```

```
enr Composition req
      Two-Operators[t1, t2, t3 / g, f]
   opns  o  :  t1  →  t3
   vars  x  :  t1
   rules
      <>  o(x)  ==>  g(f(x))
endenr
```

```
prop For-Homomorphism[t1,t2 / e,f]
   opns e  :   -      → t2
        f  :  t1 × t2 → t2
endprop

enr Homomorphism req
      For-Homomorphism[t1,t2 / e,f]
   opns hom  : Seq[t1]  → t2
   vars x    : t1
        s    : Seq[t1]
   rules
      <>      hom(nil)  ==>  e
      <>  hom(x <+ s)   ==>  x f hom(s)
endenr
```

```
prop Formal-Op[t1,t2 / f]
   opns f : t1 → t2
endprop

enr Alpha req Formal-Op[t1,t2 / f]
   opns alpha  : Seq[t1] → Seq[t2]
   vars x      : t1
        s      : Seq[t1]
   rules
      <>       alpha(nil)  ==>  nil
      <>  alpha(x <+ s)    ==>  f(x) <+
                                alpha(s)
endenr
```

La définition de l'opérateur d'homomorphisme *hom* est reprise de [9]. Dans le cas où t_1 et t_2 représentent la même sorte, cet opérateur peut être compris comme équivalent à la forme fonctionnelle d'insertion ("/") de John Backus [4], lorsque f possède un élément neutre.

```
prop For-Curry[t1,t2,t3 / f,u]
   opns f : t1 × t2 → t3
        u :   -      → t2
endprop

enr Curry req For-Curry[t1,t2,t3 / f,u]
   opns curry : t1 → t3
   vars x     : t1
   rules
      <>  curry(x)  ==>  x f u
endenr
```

```
prop For-Cond[t1,t2 / c,f1,f2]
   opns c      : t1 → Bool
        f1,f2  : t1 → t2
endprop

enr Cond req For-Cond[t1,t2 / c,f1,f2]
   opns cond  : t1 → t2
   vars x     : t1
   rules
      <>  cond(x)  ==>  if  c(x) then f1(x)
                            else f2(x)
                        endif
endenr
```

Exemple B.8 *Fonction retournant le nombre d'apparitions d'un élément dans une séquence, spécifiée à l'aide des opérateurs génériques précédents. Un opérateur formel d'égalité est bien sûr nécessaire.*

```
enr req Equality[t / eq]
   opns nb-of : t × Seq[t] → Nat
   vars x     : t
        s     : Seq[t]
   rules
      <>  nb-of(x,s)  ==>  o[hom[0,+],alpha[cond[curry[eq,x],cst[1],cst[0]]]](s)
endenr
```

Nous voyons ici l'utilisation d'une variable dans l'instanciation d'un opérateur. Remarquons également l'emploi d'entiers sous forme décimale dans des instanciations. Ils sont alors compris comme des opérateurs de profil "$- \rightarrow$ *Nat*".

B.2.2 Structures algébriques

Pour tout $n \geqslant 1$, il est possible de définir un type :

$$Sum^n.Ftype^n[t_1, \ldots, t_n\ /\ ]$$

représentant la somme disjointe de n types formels, ainsi qu'un type :

$$CP^n.Ftype^n[t_1, \ldots, t_n\ /\ ]$$

représentant le produit cartésien de n types formels.

Nous donnons dans les deux exemples ci-après les spécifications pour $n = 2$.

Exemple B.9 *Somme disjointe de deux types.*

```
prop Ftype²[t1,t2 / ]
  -- Inféré par FP2 : includes Ftype[t1 / ], Ftype[t2 / ].
endprop

type Sum² req Ftype²[t1,t2 / ]
    cons   inj_left            : t1
           inj_right           : t2
    opns   is_left, is_right   : Sum²[t1,t2]  →  Bool  -- Noter l'instanciation implicite "Sum²[t1,t2]".
           val_left            : Sum²[t1,t2]  →  t1
           val_right           : Sum²[t1,t2]  →  t2
    vars   x1                  : t1
           x2                  : t2
    rules
      <>        is_left(inj_left(x1))    ==>  true
      <>       is_left(inj_right(x2))    ==>  false
      <>       is_right(inj_left(x1))    ==>  false
      <>      is_right(inj_right(x2))    ==>  true
      <>        val_left(inj_left(x1))   ==>  x1
      <>       val_left(inj_right(x2))   ==>  !undefined-value
      <>       val_right(inj_left(x1))   ==>  !undefined-value
      <>      val_right(inj_right(x2))   ==>  x2
endtype
```

Exemple B.10 *Produit cartésien.*

Ce qui justifie l'intérêt du produit cartésien de n types, c'est qu'un terme de cette sorte représente *un* objet d'*une* sorte, et non n objets de n sortes, arguments d'une fonction n-aire. Des utilisations dans ce but figurent dans l'exemple B.3 et dans le chapitre 5.

```
type CP² req Ftype²[t1,t2 / ]
    cons   make-cp²  :  t1 × t2
    opns   ↑1        :  CP²[t1,t2]  →  t1
           ↑2        :  CP²[t1,t2]  →  t2
    vars   x1        :  t1                  --  "⟨x1,x2⟩" est une notation abrégée pour un
           x2        :  t2                  --  produit cartésien.
    rules
      <>  ↑1(⟨x1,x2⟩)  ==>  x1
      <>  ↑2(⟨x1,x2⟩)  ==>  x2
endtype
```

Exemple B.11 (Instanciation partielle) *"Carré" d'un type.*

```
type Square [t] is CP²[t,t]
endtype
```

Exemple B.12 *Propriétés d'une loi de composition interne.*

```
prop Associativity[t / f]
   opns f        : t × t  →  t
   vars x, y, z  : t
   eqns
      <>  (x f y) f z  ==  x f (y f z)
endprop
```

```
prop Commutativity[t / f]
   opns f     : t × t  →  t
   vars x, y  : t
   eqns
      <>  x f y  ==  y f x
endprop
```

Note En l'absence de parenthèses, le terme "$x\ f\ y\ f\ z$" est *parsé* comme "$(x\ f\ y)\ f\ z$".

```
prop Neutral[t / f, e]
   opns f : t × t  →  t
        e : –      →  t
   vars x : t
   eqns
      <>  e f x  ==  x
      <>  x f e  ==  x
endprop
```

```
prop Symmetrization[t / f, e, inv]
   opns f    : t × t  →  t
        e    : –      →  t
        inv  : t      →  t
   vars x    : t
   eqns
      <>  x f inv(x)  ==  e
      <>  inv(x) f x  ==  e
   includes  Neutral[t / f, e]
endprop
```

```
prop Monoid[t / f, e]
   opns f : t × t  →  t
        e : –      →  t
   includes  Associativity[t / f]
             Neutral[t / f, e]
endprop
```

```
prop Group[t / f, e, inv]
   opns f    : t × t  →  t
        e    :   –    →  t
        inv  :   t    →  t
   includes  Monoid[t / f, e]
             Symmetrization[t / f, e, inv]
endprop
```

```
model NAT⁺ is Monoid[Nat / +, 0]
endmodel
```

```
model BOOL⇔ is Group[Bool / ⇔, true, identity]
   -- Noter l'instanciation implicite de
   -- l'opérateur identity.
endmodel
```

B.2.3 Arbres

Exemple B.13 *Définition des arbres à nombre quelconque de fils, égalité et ordre lexicographique.*

```
type Tree[t]
   cons nil  : –
        <++  : t × Seq[Tree[t]]
endtype
```

```
enr req Equality[t / rel, eq]
   opns  =        : Tree[t] × Tree[t]  →  Bool
   vars  x1, x2   : t
         s1, s2   : Seq[Tree[t]]
   rules
      <>              nil = nil              ==> true
      <>              nil = (x2 <++ s2)      ==> false
      <>    (x1 <++ s1) = nil                ==> false
      <>    (x1 <++ s1) = (x2 <++ s2)        ==> if  x1 eq x2 then =.Equality[Tree[t] / =](s1, s2)
                                                     else false
                                                     --  À ce niveau, on ne sait pas encore que le
                                                     --  type Tree[t] muni de "=" est un modèle
                                                     --  de l'égalité. Noter que l'instanciation
                                                     --  explicite simplifiée "=[=](s1, s2)" est
                                                     --  également correcte.
                                                 endif
endenr
```

```
model EQTREE req Equality[t / eq] is Equality[Tree[t] / =]
endmodel
```

```
enr Linearization[t]
   opns  lin  : Tree[t]        →  Seq[t]
   vars  x    : t
         s    : Seq[Tree[t]]
   rules
      <>          lin(nil)   ==>  nil
      <>   lin(x <++ s)      ==>  x <+ hom[nil, +](alpha[lin](s))
endenr
```

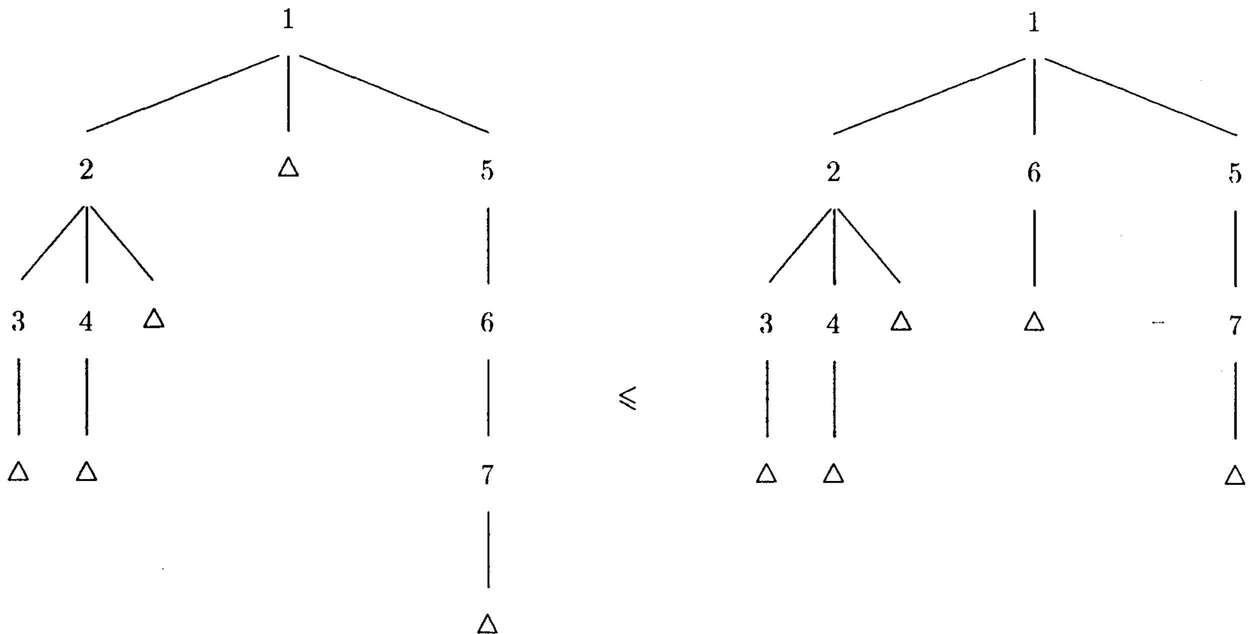
```
enr req Total-Order[t / rel, eq]
   opns  ⩽        : Tree[t] × Tree[t]  →  Bool
   vars  x1, x2   : t
         tau      : Tree[t]
         s1, s2   : Seq[Tree[t]]
   rules
      <>              nil ⩽ tau              ==> true
      <>    (x1 <++ s1) ⩽ nil                ==> false
      <>    (x1 <++ s1) ⩽ (x2 <++ s2)        ==> if  x1 eq x2 then ⩽[⩽, =](s1, s2) else x1 rel x2
                                                     --  Voir le commentaire précédent, sur
                                                     --  l'égalité des arbres.
                                                 endif
endenr
```

```
model ORDTREE req Total-Order[t / rel, eq] is Total-Order[Tree[t] / ⩽, =]
endmodel
```

Exemple B.14 *Définition d'un modèle implicite d'ordre total pour les entiers naturels, et utilisation de ce modèle pour le calcul d'une instanciation implicite en vue d'ordonner deux arbres quelconques d'entiers.*

model $INCNAT$ **is** *Total-Order*[*Nat* / $\leqslant, =$]
endmodel

$1 <\!\!++\ [2 <\!\!++\ [3 <\!\!++\ \mathit{nil}, 4 <\!\!++\ \mathit{nil}, \mathit{nil}], \mathit{nil}, 5 <\!\!++\ [6 <\!\!++\ [7 <\!\!++\ \mathit{nil}]]] \leqslant$
$1 <\!\!++\ [2 <\!\!++\ [3 <\!\!++\ \mathit{nil}, 4 <\!\!++\ \mathit{nil}, \mathit{nil}], 6 <\!\!++\ \mathit{nil}, 5 <\!\!++\ [7 <\!\!++\ \mathit{nil}]]$ \$
true



Exemple B.15 *Définition d'utilitaires divers pour un type ordonné.*

```
enr req Total-Order[t / rel, eq]
   opns  ord?          : Seq[t]  →  Bool
         min2, max2    : t × t   →  t
         min, max      : Seq[t]  →  t
   vars  x, y, z       : t
         s             : Seq[t]
   rules
      <>      ord?(nil)   ==>  true
      <>  ord?(x <+ s)    ==>  if  nil?(s) then true
                                   elsif ord?(s) then x rel head(s)
                                   else false
                               endif
      <>     min2(x, y)   ==>  if  x rel y then x else y
                               endif
      <>     max2(x, y)   ==>  if  x rel y then y else x
                               endif
      <>       min(nil)   ==>  !undefined-value
      <>   min(x <+ s)    ==>  hom[x, min2](s)
                               --  Noter l'emploi de l'opérateur hom (cf. exemple B.7) et
                               --  l'utilisation d'une variable dans l'instanciation.
      <>       max(nil)   ==>  !undefined-value
      <>   max(x <+ s)    ==>  hom[x, max2](s)
endenr
```

Exemple B.16 *Définition des arbres binaires ordonnés, et application au tri par arbre.*

```
type O-Tree req Total-Order[t / rel, eq]
  cons  nil           : -
        ⟨#⟩           : O-Tree[t] × t × O-Tree[t]
  opns  tree-insert   : t × O-Tree[t]          → O-Tree[t]
        nil?          : O-Tree[t]              → Bool
        lin           : O-Tree[t]              → Seq[t]     -- Linéarisation.
  vars  x, y          : t
        s1, s2        : Seq[t]
        tau1, tau2    : O-Tree[t]
  preds
    <>  ⟨#⟩(tau1, x, tau2)  |  not  ((if nil?(tau1) then true else max(lin(tau)) rel x
                                     endif) and
                                    (if nil?(tau2) then true else x rel min(lin(tau2))
                                     endif))
                                        ==>  !wrong-tree
  rules
                                     -- Alors que "⟨#⟩" est un constructeur assujetti
                                     -- à une précondition dans laquelle on vérifie
                                     -- que la disposition des éléments est conforme à
                                     -- la relation d'ordre en paramètre, l'opérateur
                                     -- tree-insert permet de ranger n'importe quel
                                     -- élément à la place qui lui revient.
    <>                tree-insert(x, nil)  ==>  ⟨#⟩(nil, x, nil)
    <>  tree-insert(y, ⟨#⟩(tau1, x, tau2))  ==>  if  x rel y then ⟨#⟩(tree-insert(y, tau1), x, tau2)
                                                  else ⟨#⟩(tau1, x, tree-insert(y, tau2))
                                                endif
    <>                         nil?(nil)  ==>  true
    <>          nil?(⟨#⟩(tau1, x, tau2))  ==>  false
    <>                          lin(nil)  ==>  nil
    <>           lin(⟨#⟩(tau1, x, tau2))  ==>  lin(tau1) + (x <+ lin(tau2))
endtype
```

```
enr Tree-Sort req Total-Order[t / rel, eq]
  opns  tree-sort     : Seq[t]  →  Seq[t]
  priv  do-tree-sort  : Seq[t]  →  Seq[t]
  vars  x             : t
        s             : Seq[t]
  rules
    <>            tree-sort(s)  ==>  lin(do-tree-sort(s))
    <>       do-tree-sort(nil)  ==>  nil
    <>  do-tree-sort(x <+ s)    ==>  tree-insert(x, do-tree-sort(s))
endenr
```

B.2.4 Utilisation d'exceptions

Exemple B.17 *Type Bounded-Stack des piles bornées génériques.*

```
prop For-Bounded-Stack[t / p]
  opns  p  :  -  →  Nat
endprop
```

```
type Bounded-Stack req For-Bounded-Stack[t / p]
   cons  empty  : -
         push   : t × Bounded-Stack[t]
   opns  top    : Bounded-Stack[t]    → t
         pop    : Bounded-Stack[t]    → Bounded-Stack[t]
         depth  : Bounded-Stack[t]    → Nat
   vars  x      : t
         s      : Bounded-Stack[t]
   prcds
      <>  x push s  |  depth(s) ⩾ p  ==>  !too-deep
   rules
      <>       top(empty)  ==>  !too-deep
      <>    top(x push s)  ==>  x
      <>       pop(empty)  ==>  !too-deep
      <>    pop(x push s)  ==>  s
      <>     depth(empty)  ==>  0
      <>  depth(x push s)  ==>  succ(depth(s))
endtype
```

Exemple B.18 *Si nous considérons un espace mémoire comme une séquence de piles bornées, l'allocation d'une nouvelle pile dans le but d'y ranger un élément supplémentaire peut dès lors s'effectuer par récupération du terme exceptionnel "!too-deep".*

```
type Page req For-Bounded-Stack[t / p] is Seq[Bounded-Stack[t]]
endtype
```

```
enr Alloca req For-Bounded-Stack[t / p]
   opns  init     : -             → Page[t]
         new-elt  : t × Page[t]   → Page[t]
   vars  x        : t
         pp       : Page[t]
   rules
      <>            init  ==>  [empty]
      <>  new-elt(x, pp)  ==>  rec  (x push head(pp)) <+ tail(pp)
                                    when !too-deep then (x push empty) <+ pp
                               endrec
endenr
```

B.2.5 Définition des caractères et des chaînes de caractères par transfert de structure

Exemple B.19 *Définition d'un isomorphisme à partir d'un opérateur de conversion.*

```
prop Iso-and-Conv[t1, t2, t3 / f, c]
   opns  f  :  t2 × t2  →  t3
         c  :     t1    →  t2
endprop
```

```
enr Isomorphism req Iso-and-Conv[t1, t2, t3 / f, c]
   opns  iso  :  t1 × t1  →  t3
   vars  x, y :  t1
   rules
      <>  iso(x, y)  ==>  c(x) f c(y)
endenr
```

Exemple B.20 *Type Char des caractères, définis par leur code.*

```
type Char
  cons  make-char    : Nat
  opns  code-char    : Char          →  Nat
        =,<,⩽,>,⩾    : Char × Char   →  Bool
  vars  i            : Nat
        ch1, ch2     : Char
  rules
    <>  code-char(make-char(i))  ==>  i
    <>  ch1  =  ch2  ==>  iso.Iso-and-Conv[Char, Nat, Bool / =, code-char](ch1, ch2)
    <>  ch1  <  ch2  ==>  iso.Iso-and-Conv[Char, Nat, Bool / <, code-char](ch1, ch2)
    <>  ch1  ⩽  ch2  ==>  iso.Iso-and-Conv[Char, Nat, Bool / ⩽, code-char](ch1, ch2)
    <>  ch1  >  ch2  ==>  iso.Iso-and-Conv[Char, Nat, Bool / >, code-char](ch1, ch2)
    <>  ch1  ⩾  ch2  ==>  iso.Iso-and-Conv[Char, Nat, Bool / ⩾, code-char](ch1, ch2)
endtype
```

```
model EQCHAR is Equality[Char / =]
endmodel
```

Le *parser* accepte également les caractères sous une forme plus familière : 'a, 'b, ..., 'RETURN, ...

Exemple B.21 *Type String des chaînes de caractères. Les types String et Seq[Char] étant isomorphes, les définitions de l'égalité, de l'ordre sur les chaînes, et de la concaténation s'effectuent par transfert des opérateurs correspondants dans Seq[Char].*

```
type String
  cons  empty-string      : –
        <+                : Char × String
  opns  coerce-string     : Seq[Char]            →  String
        coerce-char-list  : String               →  Seq[Char]
        sub-string        : String × Nat × Nat   →  String
  vars  i, j              : Nat
        ch                : Char
        str               : String
        s                 : Seq[Char]
  rules
    <>                    coerce-string(s)  ==>  hom[empty-string, <+](s)
    <>  coerce-char-list(empty-string)      ==>  nil
    <>      coerce-char-list(ch <+ str)     ==>  ch <+ coerce-char-list(str)
    <>              sub-string(str, i, j)   ==>
                                   rec  coerce-string(sub-seq(coerce-char-list(str), i, j))
                                        when !undefined-value then !undefined-string
                                   endrec
endtype
```

enr
opns $=, <, \leqslant, >, \geqslant$: $String \times String \rightarrow Bool$
$+$: $String \times String \rightarrow String$
vars str_1, str_2 : $String$
rules
<> $str_1 \ = \ str_2$ ==>
iso.Iso-and-Conv[*String*, *Seq*[*Char*], *Bool* / =, *coerce-char-list*](str_1, str_2)
-- Noter l'instanciation implicite de l'opérateur "=", utilisant la déclaration de modèle
-- *EQCHAR*.
<> $str_1 \ < \ str_2$ ==> ($str_1 \leqslant str_2$) *and not*($str_1 = str_2$)
<> $str_1 \ \leqslant \ str_2$ ==>
iso.Iso-and-Conv[*String*, *Seq*[*Char*], *Bool* / $\leqslant[\leqslant, =]$, *coerce-char-list*](str_1, str_2)
<> $str_1 \ > \ str_2$ ==> *not*($str_1 \leqslant str_2$)
<> $str_1 \ \geqslant \ str_2$ ==> *not*($str_1 < str_2$)
<> $str_1 \ + \ str_2$ ==>
coerce-string
(*iso.Iso-and-Conv*[*String*, *Seq*[*Char*], *Seq*[*Char*] / +, *coerce-char-list*](str_1, str_2))
endenr

model *EQSTRING* **is** *Equality*[*String* / =]
endmodel

Dès lors qu'existe le type *Seq*[*Char*], la principale raison d'être du type *String* est l'interface permettant de lire et d'écrire les chaînes entre doubles guillemets ("*FP2*"). Pour insérer un double guillemet à l'intérieur d'une chaîne, il suffit de le doubler : "For ""ever"" and ""a"" day".

B.3 Exemples de codes générés par le compilateur FP2

Nous donnons ci-après les codes générés par le compilateur FP2 pour les opérateurs d'égalité et d'ordre lexicographique sur les séquences, et sur les arbres à nombre quelconques de fils.

Nous rappelons encore une fois que — cf. chapitre 2 — :

- les booléens de FP2 sont compilés sous leur forme Lisp la plus "naturelle" : *true* en `t` et *false* en `nil` ;
- les séquences et les arbres sont représentés par des listes Common Lisp (cf. §2.1.2.2 et tableau p. 86) : ainsi, les deux opérateurs FP2 notés *nil* (la liste vide et l'arbre vide) sont tous deux compilés en le symbole `nil` de Lisp, quant aux opérateurs "<+" et "<++", ils sont tous deux compilés en la fonction `cons` ;
- par convention, la variable `fm` représente l'argument supplémentaire qui regroupe la forme compilée des opérateurs du modèle exigé (cf. §2.2.2).

```
(defun =S (x y fm)
  (if x
      (if y
          (if (funcall (first fm) (car x) (car y))
              (=S (cdr x) (cdr y) fm)))
      (endp y)))
```

```
(defun <=S (x y fm)
  (if x
      (if y
          (if (funcall (second fm) (car x) (car y))
              (<=S (cdr x) (cdr y) fm)
              (funcall (first fm) (car x) (car y))))
      t))

(defun =T (x y fm)
  (if x
      (if y
          (if (funcall (first fm) (car x) (car y))
              (=S (cdr x) (cdr y) (list #'(lambda (z0 z1) (=T z0 z1 fm))))))
      (endp y)))

(defun <=T (x y fm)
  (if x
      (if y
          (if (funcall (second fm) (car x) (car y))
              (<=S (cdr x) (cdr y)
                   (list #'(lambda (z0 z1) (<=T z0 z1 fm))
                         #'(lambda (z2 z3) (=T z2 z3 (list (second fm))))))
              (funcall (first fm) (car x) (car y))))
      t))
```

Annexe C

Linéarisation d'un système de réécriture

Dans cette annexe, nous allons définir une relation d'équivalence entre systèmes de réécriture, et des conditions sous lesquelles nous pouvons ramener tout système de réécriture à un système de réécriture linéaire à gauche équivalent au sens de cette relation. La preuve est suivie de deux exemples. Nous terminons par un résultat de confluence découlant lui aussi de cette relation d'équivalence.

Notre but principal est de parachever la preuve du théorème 4.3, néanmoins, cette annexe peut se lire indépendamment du chapitre 4.

C.1 Quelques définitions techniques

Définition C.1 *Soit s une sorte. s est une* **sorte avec égalité** *s'il est possible de spécifier un opérateur eq, de profil $s \times s \rightarrow Bool$, au moyen d'un système de réécriture convergent et linéaire à gauche, et ceci de façon complète, c'est-à-dire :*

$$(\forall t_1, t_2 \text{ termes de sorte } s), eq(t_1, t_2) \longrightarrow true \vee eq(t_1, t_2) \longrightarrow false$$

Dans le §C.1, nous allons considérer que la sorte s admet m constructeurs constants $(c_i)_{1 \leqslant i \leqslant m}$ et n constructeurs d'arité non nulle $(g_j)_{1 \leqslant j \leqslant n}$.

Proposition C.2 *En l'absence d'équations entre ses constructeurs, une sorte est une sorte avec égalité, dès lors que toutes les autres sortes qui apparaissent dans les domaines des constructeurs sont des sortes avec égalité.*

eq peut se définir par :

$$\begin{array}{rcl} eq(c_1, c_1) & \longrightarrow & true \\ eq(c_1, c_2) & \longrightarrow & false \\ & \vdots & \\ eq(g_1(x_1, \ldots, x_{\mathrm{arity}(g_1)}), g_1(y_1, \ldots, y_{\mathrm{arity}(g_1)})) & \longrightarrow & \displaystyle\bigwedge_{1 \leqslant i \leqslant \mathrm{arity}(g_1)} eq(x_i, y_i) \\ & \vdots & \end{array} \tag{C.1}$$

$$eq(g_1(x_1, \ldots, x_{\text{arity}(g_1)}), g_2(y_1, \ldots, y_{\text{arity}(g_2)})) \longrightarrow false$$
$$\vdots$$

Note Dans la règle (C.1), *eq* doit être compris comme un symbole surchargé, défini sur $s_i \times s_i \rightarrow Bool$, de façon identique pour toutes les sortes s_i qui apparaissent dans le domaine de g_1.

Bien sûr, toutes les sortes de FP2 sont des sortes avec égalité. En cas d'équations entre constructeurs, non seulement la spécification d'un tel opérateur est moins triviale, mais elle peut être *impossible* sans l'ajout d'opérateurs auxiliaires (à titre d'exemple, Hubert Comon rappelle dans [43] que l'on ne peut pas spécifier l'égalité entre entiers relatifs sans opérateur auxiliaire).

Proposition C.3 ([151, 43]) *Lorsque les équations entre constructeurs sont orientables en un système de réécriture, il est possible de construire une grammaire qui permet d'engendrer toutes les formes normales fermées d'une sorte.*

Dans ce cas, on peut toujours spécifier l'égalité, éventuellement à l'aide d'un opérateur auxiliaire de normalisation.

Remarque Par contre, à notre connaissance, lorsqu'on se trouve en présence d'équations non orientables (par exemple, une équation de commutativité), il n'existe pas de méthode générale pour spécifier linéairement un opérateur d'égalité. Toujours à notre connaissance, il n'existe pas non plus d'exemple de sorte pour laquelle il a été prouvé que cette spécification est impossible.

Définition C.4 Scinder une variable *de sorte s dans un terme t, c'est générer autant de termes que de constructeurs de sorte s, en ayant remplacé dans ces termes la variable par une expression formée à l'aide d'un constructeur et éventuellement de nouvelles variables.*

Plus formellement :

$$split(x, t) = \{\sigma(t) \mid \sigma \in \{[x \mapsto c_i]\}_{1 \leqslant i \leqslant m} \cup \{[x \mapsto g_j(x_{j,1}, \ldots, x_{j,\text{arity}(g_j)})]\}_{1 \leqslant j \leqslant n}\}$$

Dans toute cette annexe, nous supposerons que tous les systèmes de réécriture sont définis sur la même signature visible.

Définition C.5 *Deux systèmes de réécriture $\mathcal{R}$ et $\mathcal{S}$, éventuellement conditionnels, seront dits* **équivalents par rapport aux constructeurs** *si et seulement si :*

(i) toute forme normale[85] *d'un terme fermé quelconque, obtenue au moyen de $\mathcal{R}$ (resp. $\mathcal{S}$) et pouvant s'exprimer uniquement à l'aide de constructeurs, peut également s'obtenir au moyen de $\mathcal{S}$ (resp. $\mathcal{R}$) ;*

(ii) tout terme fermé tel qu'une de ses formes normales obtenues au moyen de $\mathcal{R}$ (resp. $\mathcal{S}$) ne peut pas s'exprimer uniquement au moyen de constructeurs vérifie la même propriété pour $\mathcal{S}$ (resp. $\mathcal{R}$).

Remarquons que :

1. la condition (ii) exige l'impossibilité commune d'exprimer uniquement à l'aide de constructeurs les deux formes normales obtenues au moyen de $\mathcal{R}$ et de $\mathcal{S}$, mais n'exige pas leur égalité,

[85] *Toute* forme normale, parce que les systèmes ne sont pas supposés confluents.

2. cette relation est bel et bien une relation d'équivalence entre systèmes de réécriture.

Cette définition revient à ne considérer que les termes irréductibles composés uniquement de constructeurs, les autres étant soit mal formés, soit — et c'est la solution la plus naturelle compte tenu du contexte dans lequel nous nous plaçons au chapitre 4 — réécrits en un opérateur d'exception.

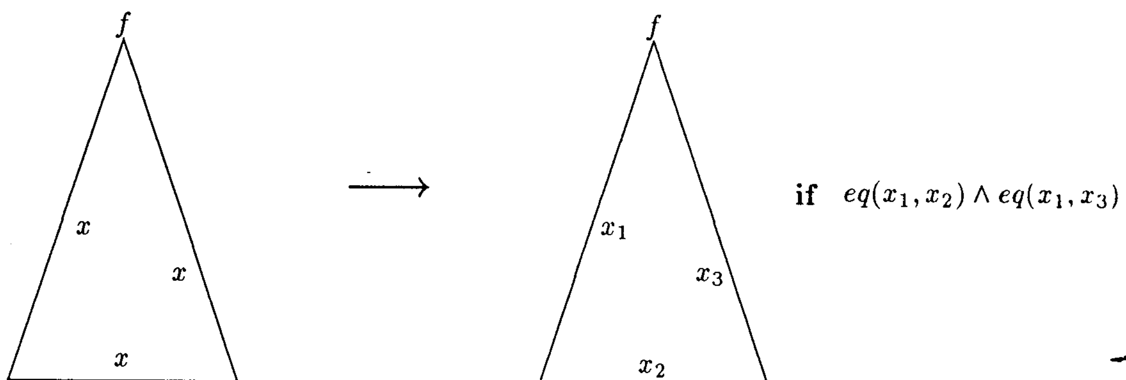
C.2 Linéarisation d'un système de réécriture

Théorème C.6 *Pour tout système de réécriture $\mathcal{R}$, opérant sur des termes dont les sortes sont des sortes avec égalité, il existe un système de réécriture $\mathcal{R}^\natural$, qui est linéaire à gauche et équivalent à $\mathcal{R}$ par rapport aux constructeurs.*

Dans la description de la transformation d'un système de réécriture $\mathcal{R}$ en un tel système de réécriture $\mathcal{R}^\natural$ et sa preuve, nous traitons le cas d'un seul opérateur non constructeur f, la méthode se généralisant aisément aux systèmes de réécriture définissant plusieurs opérateurs.

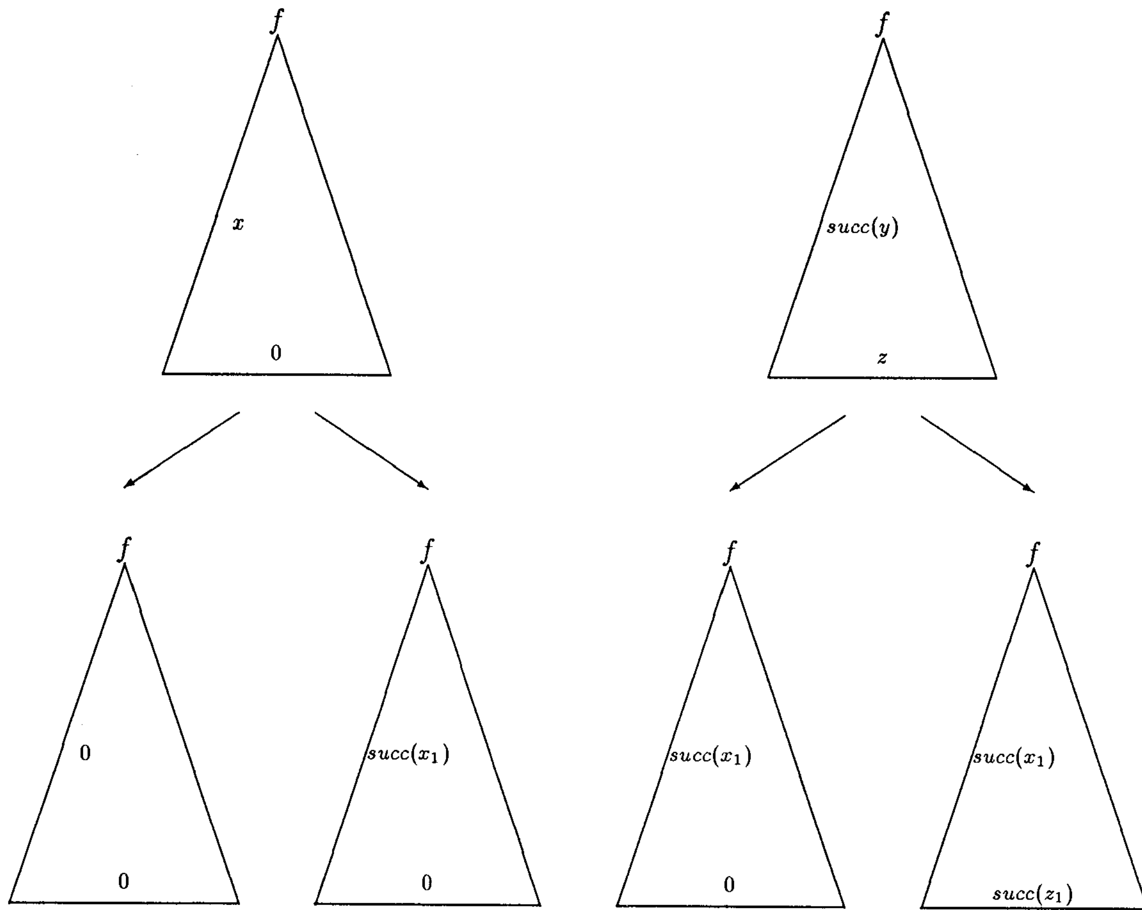
Nous allons commencer par transformer $\mathcal{R}$ en un système de réécriture conditionnel. On indice par des entiers naturels non nuls et croissants les différentes occurrences d'une variable en partie gauche. La prémisse est formée par la conjonction des termes de la forme $eq(z_1, z_{p'})$, avec $1 < p' \leqslant p$, pour toute variable z apparaissant p fois ($p > 1$) dans la partie gauche. D'après l'hypothèse, la spécification de tels opérateurs eq au moyen de règles linéaires à gauche est possible pour toute sorte. Quant aux variables de la partie droite, nous les indiçons par "1".

En représentant schématiquement les parties gauches :



Nous considérons ensuite les parties gauches superposables (unifiables) et nous en scindons les variables jusqu'à ce que deux quelconques de ces parties gauches soient ou bien non unifiables, ou bien identiques à un renommage près. Nous en profitons pour :

- simplifier les prémisses, et éventuellement enlever les règles dont la prémisse est trivialement fausse,
- effectuer des renommages de telle sorte que deux quelconques des parties gauches soient ou bien non unifiables, ou bien égales syntaxiquement, c'est-à-dire identiques y compris pour les noms des variables.



Nous nommons $\mathcal{R}^b$ le système de réécriture conditionnel obtenu à ce niveau de la transformation. Rappelons qu'à ce niveau, deux parties gauches quelconques sont soit non unifiables, soit syntaxiquement égales.

Nous partitionnons $\mathcal{R}^b$ en des sous-ensembles que nous appellerons **sous-ensembles de factorisation** : ces sous-ensembles sont tels que toutes les parties gauches des règles d'un sous-ensemble sont identiques. Si toutes les règles de réécriture d'un sous-ensemble de factorisation ont une prémisse trivialement vraie, ces règles sont reprises sans modification dans $\mathcal{R}^\natural$. Dans le cas contraire, pour un sous-ensemble de factorisation de $\mathcal{R}^b$, soit q le nombre de tests (sous-termes d'une conjonction, formés à l'aide d'opérateurs eq) contenus dans les prémisses, et qui mettent en jeu des paires de variables distinctes. Nous définissons un ordre lexicographique et *ordonnons* les q paires de variables utilisées dans ces prémisses. Par conséquent, chaque prémisse γ de ce sous-ensemble peut être définie au moyen d'une fonction :

$$\ell_\gamma : \{1, \ldots, q\} \rightarrow \{true, false\}$$

telle que $\ell_\gamma(i)$ soit vrai si et seulement si le test de rang i est présent dans la prémisse γ. On introduit alors un opérateur auxiliaire *caché* par sous-ensemble. Si l'opérateur f, racine des parties gauches, a le profil :

$$s_1 \times \cdots \times s_n \rightarrow s$$

alors l'opérateur auxiliaire f' a le profil :

$$s_1 \times \cdots \times s_n \times \overbrace{Bool \times \cdots \times Bool}^{q \text{ fois}} \rightarrow s$$

Nous construisons alors une règle dont la partie gauche est la partie gauche commune des règles du sous-ensemble et dont la partie droite est formée au moyen de l'opérateur auxiliaire : les n premiers arguments de f' (n étant l'arité de f) sont identiques aux n arguments de f en partie gauche, et les arguments suivants sont constitués par les tests extraits des prémisses, dans l'ordre choisi pour le sous-ensemble considéré.

Pour terminer la transformation, nous reprenons chaque règle conditionnelle du sous-ensemble de factorisation, et donnons une règle équivalente construite avec l'opérateur auxiliaire en partie gauche. Les n premiers arguments sont les arguments de f dans la partie gauche du sous-ensemble de $\mathcal{R}^\flat$, et les q suivants sont construits d'après ℓ_γ. Pour toute place i, avec $n+1 \leqslant i \leqslant n+q$:

- si ℓ_γ est vrai, le terme d'occurrence i est "*true*",
- sinon, une nouvelle variable de sorte *Bool* est créée.

La partie droite est celle de la règle conditionnelle. Les variables créées ne se situant qu'en partie gauche, toutes les variables présentes en partie droite sont présentes en partie gauche.

Ainsi est obtenu le système $\mathcal{R}^\natural$. Il est assez facile de voir que $\mathcal{R}^\flat$ est strictement équivalent à $\mathcal{R}$ par construction. Ce qui nous reste à prouver, c'est que $\mathcal{R}^\natural$ est équivalent à $\mathcal{R}^\flat$ par rapport aux constructeurs.

Soit $\mathcal{S}$ un système de réécriture. Notons par $Aux(t)$ la proposition "le terme t contient des opérateurs auxiliaires cachés" et adoptons la notation suivante, traduisant la notion de "réécriture tant que le résultat contient des opérateurs cachés" :

$$t_1 \overset{\sim}{\longrightarrow}_{\mathcal{S}} t_2 \overset{\text{déf}}{\Longleftrightarrow} \left\{ \begin{array}{c} (t_1 \longrightarrow_{\mathcal{S}} t_2 \wedge \neg Aux(t_2)) \\ \vee \\ (\exists t'_1), (t_1 \longrightarrow_{\mathcal{S}} t'_1 \wedge Aux(t'_1) \wedge t'_1 \overset{\sim}{\longrightarrow}_{\mathcal{S}} t_2) \end{array} \right.$$

Lemme C.7 *Soient $\mathcal{R}$ et $\mathcal{S}$, deux systèmes de réécriture, éventuellement conditionnels. Notons T_{Σ_Γ} l'ensemble des termes fermés s'exprimant uniquement à l'aide de constructeurs. Si, pour tous $t_1, t_2 \in T_\Sigma$, et tout $t_3 \in T_{\Sigma_\Gamma}$:*

$$(t_1 \longrightarrow_{\mathcal{R}} t_2) \iff (t_1 \overset{\sim}{\longrightarrow}_{\mathcal{S}} t_2) \tag{C.2}$$

$$(t_3 \textit{ irréductible par } \mathcal{R}) \iff (t_3 \textit{ irréductible par } \mathcal{S}) \tag{C.3}$$

alors toute forme normale d'un terme t de T_Σ, obtenue au moyen de $\mathcal{R}$ (resp. $\mathcal{S}$) et appartenant à T_{Σ_Γ} peut s'obtenir au moyen de $\mathcal{S}$ (resp. $\mathcal{R}$).

Preuve du lemme C.7 Soient $t \in T_\Sigma$ et $u \in T_{\Sigma_\Gamma}$ une forme normale de t, obtenue au bout de p applications de règles de $\mathcal{R}$. Cela signifie que :

$$(\exists (u_i)_{1 \leqslant i \leqslant p}), t \longrightarrow_{\mathcal{R}} u_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} u_p = u$$

D'après (C.2) :

$$t \overset{\sim}{\longrightarrow}_{\mathcal{S}} u_1 \overset{\sim}{\longrightarrow}_{\mathcal{S}} \cdots \overset{\sim}{\longrightarrow}_{\mathcal{S}} u_p = u$$

u est un terme de T_{Σ_Γ} irréductible par $\mathcal{R}$: d'après (C.3), u est irréductible par $\mathcal{S}$, c'est donc une forme normale de t qui peut s'obtenir au moyen de $\mathcal{S}$.

Dans le sens contraire, soit $v \in T_{\Sigma_\Gamma}$ une forme normale de t obtenue par $\mathcal{S}$. v ne contenant pas d'opérateurs auxiliaires, il est possible de décomposer les réécritures successives de t en une suite de relations "$\overset{\sim}{\longrightarrow}_{\mathcal{S}}$". Ensuite, on prend en compte de façon analogue les hypothèses (C.2) et (C.3) et l'on obtient le résultat.

Preuve du théorème C.6 à l'aide du lemme C.7 Rappelons que nous considérons une signature avec un seul opérateur f non constructeur défini non linéairement, la généralisation au cas de plusieurs opérateurs étant aisée. Rappelons également que seules ont été modifiées, lors du passage de $\mathcal{R}$ à $\mathcal{R}^\natural$, les règles préfixées par f. Par conséquent, un terme qui s'exprime uniquement en fonction de constructeurs, s'il est irréductible par $\mathcal{R}$, l'est également par $\mathcal{R}^\flat$ et par $\mathcal{R}^\natural$: (C.3) est vérifiée.

Étudions (C.2) en considérant un terme fermé t_1 dont au moins une occurrence est préfixée par f.

- $\boxed{t_1 \longrightarrow_{\mathcal{R}^\flat} t_2}$

  Si la règle appliquée ne comporte pas de prémisse, elle est reprise dans $\mathcal{R}^\natural$, et donc $t_1 \longrightarrow_{\mathcal{R}^\natural} t_2$. Traitons le cas d'une règle conditionnelle. Par définition de la réécriture conditionnelle :

$$(\exists\tau \in occ(t_1)), (\exists(l \longrightarrow r \text{ if } c) \in \mathcal{R}^\flat), l \overset{\sigma}{\lesssim} t_1/\tau \wedge \sigma(c) \longrightarrow true \wedge t_2 = t_1[\tau \leftarrow \sigma(r)]$$

  (Remarquons qu'en cas d'équations sur les constructeurs, le filtre σ est calculé *modulo* ces équations.)

  Le terme l est de la forme $f(u_1, \ldots, u_n)$, et est tel que :

$$(\forall i, 1 \leqslant i \leqslant n), u_i \overset{\sigma}{\lesssim} i.\tau \tag{C.4}$$

  Puisqu'on regroupe, lors du passage de $\mathcal{R}^\flat$ à $\mathcal{R}^\natural$, les règles de $\mathcal{R}^\flat$ d'après leur partie gauche, il s'ensuit qu'étant donné une règle de $\mathcal{R}^\flat$, il existe nécessairement une règle unique dans $\mathcal{R}^\natural$ qui admet même partie gauche. Si la prémisse de la règle de $\mathcal{R}^\flat$ n'est pas trivialement vraie, la partie droite de la règle correspondante dans $\mathcal{R}^\natural$ est de la forme :

$$r_0 = f'(u_1, \ldots, u_n, \overbrace{u_{n+1}, \ldots, u_{n+q}}^{\text{termes de sorte } Bool}), \text{ d'où l'on pose } t_1' = t_1[\tau \leftarrow \sigma(r_0)]$$

  f' étant un opérateur auxiliaire. Par construction, toutes les règles du sous-ensemble de factorisation qui sont préfixées par f' vérifient (C.4). Les tests d'égalité restent à examiner.

  c peut se définir par :

$$c = \bigwedge_{j \in \ell_c^{-1}(\{true\})} eq_j(x_{j,1}, x_{j,2})$$

  ($\ell_c^{-1}(A)$ désignant l'image réciproque de l'ensemble A par ℓ_c.) D'où :

$$\sigma(c) \longrightarrow true \iff (\forall j \in \ell_c^{-1}(\{true\})), eq_j(\sigma(x_{j,1}), \sigma(x_{j,2})) \longrightarrow true$$

  Par construction, il existe une règle $(l' \longrightarrow r')$ de $\mathcal{R}^\natural$ telle que :

$$\begin{array}{rl} (\forall j \in \ell_c^{-1}(\{true\})), & l'/(n+j) = true \\ (\forall j \in \{1, \ldots, q\} \setminus \ell_c^{-1}(\{true\})), & l'/(n+j) \text{ est une variable} \end{array}$$

Puisque les n premiers arguments d'une règle dont la partie gauche est préfixée par f' sont les mêmes que les n arguments de l'unique partie gauche, préfixée par f, de la règle du sous-ensemble de factorisation, il suffit d'étendre σ en une substitution σ' qui associe une image aux variables de sorte *Bool* qui ont été introduites pour les sous-termes d'occurrences $n+1$ à $n+q$. Ces variables sont situées aux occurrences $n+j$, pour $j \in \{1,\ldots,q\} \setminus \ell_c^{-1}(\{true\})$, d'où :

$$\sigma' = \sigma \uplus (\biguplus_{j \in \{1,\ldots,q\} \setminus \ell_c^{-1}(\{true\})} [l'/(n+j) \mapsto t_1'/i.\tau])$$

("$\uplus$" désignant ici l'union disjointe des graphes.) Dès lors :

$$(\forall i, 1 \leqslant i \leqslant n+q), l'/i \overset{\sigma'}{\precsim} t_1'/i.\tau$$

l' et t_1' ayant même racine : $l' \overset{\sigma'}{\precsim} t_1'/\tau$ et la règle $(l' \longrightarrow r')$ peut être appliquée. Par conséquent :

$$t_1' \longrightarrow_{\mathcal{R}^\natural} t_2' = t_1'[\tau \leftarrow \sigma'(r')]$$

Par construction, $r' = r$, d'où $\sigma'(r') = \sigma(r)$. De plus, pour tous termes u_0, u_1, u_2, on a :

$$(u_0[\tau \leftarrow u_1])[\tau \leftarrow u_2] = u_0[\tau \leftarrow u_2]$$

Finalement :

$$\begin{aligned} t_2' &= t_1'[\tau \leftarrow \sigma(r)] \\ &= (t_1[\tau \leftarrow \sigma(r_0)])[\tau \leftarrow \sigma(r)] \\ &= t_1[\tau \leftarrow \sigma(r)] \\ &= t_2 \end{aligned}$$

$$t_1 \longrightarrow_{\mathcal{R}^\natural} t_1' \longrightarrow_{\mathcal{R}^\natural} t_2$$

$$t_1 \overset{\sim}{\longrightarrow}_{\mathcal{R}^\natural} t_2 \quad \square$$

- t_1 irréductible par $\mathcal{R}^\flat$

Explicitons complètement l'hypothèse :

$$(\forall \tau \in occ(t_1)), (\forall (l \longrightarrow r \text{ if } c) \in \mathcal{R}^\flat), \left\{ \begin{array}{lr} (l \not\precsim t_1/\tau) & \text{(C.5)} \\ \vee & \\ (l \overset{\sigma}{\precsim} t_1/\tau \wedge \neg(\sigma(c) \longrightarrow true)) & \text{(C.6)} \end{array} \right.$$

Les parties gauches préfixées par f étant identiques dans $\mathcal{R}^\flat$ et dans $\mathcal{R}^\natural$, il s'ensuit que si t_1 vérifie (C.5), alors t_1 est irréductible par $\mathcal{R}^\natural$.

Si t_1 vérifie (C.6), t_1 est tel que :

$$(\forall \tau \in occ(t_1)), (\forall (l \longrightarrow r \text{ if } c) \in \mathcal{R}^\flat), l \overset{\sigma}{\precsim} t_1/\tau \Longrightarrow \sigma(r) \longrightarrow false$$

Nous venons d'utiliser que c est une conjonction de termes formés avec des opérateurs d'égalité. D'après l'hypothèse, tous ces opérateurs de codomaine *Bool* sont complètement définis par un système de réécriture nœthérien. Par conséquent :

$$\neg(\sigma(c) \longrightarrow true) \Longleftrightarrow (\sigma(c) \longrightarrow false)$$

S'il existe σ tel que $l \overset{\sigma}{\lesssim} t_1/\tau$, un raisonnement semblable au cas précédent nous permet de déduire qu'il existe dans $\mathcal{R}^\natural$ une règle de même partie gauche l. D'autre part :

$$\sigma(c) \longrightarrow false \Longleftrightarrow (\exists j \in \ell_c^{-1}(\{true\})), eq_j(\sigma(x_{j,1}), \sigma(x_{j,2})) \longrightarrow false$$

Toutes les parties gauches l' des règles du sous-ensemble de factorisation correspondant à la règle conditionnelle ($l \longrightarrow r$ if c), et préfixées par l'opérateur auxiliaire sont telles que :

$$(\forall j \in \ell_c^{-1}(\{true\})), l'/(n+j) = true$$

Par conséquent :

$$(\exists j \in \ell_c^{-1}(\{true\})), l'/(n+j) \not\lesssim eq_j(\sigma(x_{j,1}), \sigma(x_{j,2}))$$

Si une règle de réécriture de $\mathcal{R}^\flat$ échoue à cause de sa prémisse, aucune règle du sous-ensemble de factorisation correspondant dans $\mathcal{R}^\natural$ ne possède de règle applicable préfixée par l'opérateur auxiliaire. Globalement, si toutes les règles de $\mathcal{R}^\flat$ échouent, le terme est soit irréductible par $\mathcal{R}^\natural$, soit réductible en un terme irréductible préfixé par un opérateur auxiliaire. □

- Les réciproques se démontrent de manière analogue et à partir des mêmes termes, les opérateurs auxiliaires ajoutés dans $\mathcal{R}^\natural$ étant cachés.

Corollaire C.8 *Pour tout opérateur f, complètement défini par un système de réécriture $\mathcal{R}$, c'est-à-dire tel que toute forme normale d'un terme préfixé par f peut s'exprimer uniquement en fonction de constructeurs, il existe un système de réécriture $\mathcal{R}^\flat$ équivalent et linéaire à gauche.*

Ce corollaire s'obtient par la seule utilisation de l'axiome (i) de la définition C.5. Dans ce cas, la forme normale de chaque terme accessible par l'emploi de f et préfixé par un opérateur auxiliaire caché s'exprime uniquement en fonction de constructeurs. Par conséquent, il est à remarquer, dans ce cas, que l'introduction de ces opérateurs ne perturbe pas la propriété d'existence éventuelle d'une algèbre initiale satisfaisant la présentation.

Donnons à présent deux exemples d'opérateurs définis non linéairement :

Exemple C.9 *Soit f_0 un opérateur de majorité :*

$$\begin{array}{lcl} f_0(x,x,y) & \longrightarrow & x \\ f_0(x,y,x) & \longrightarrow & x \\ f_0(y,x,x) & \longrightarrow & x \end{array}$$

Le système de réécriture conditionnelle est :

$$\begin{array}{lcl} f_0(z_1,z_2,z_3) & \longrightarrow & z_1 \text{ if } eq(z_1,z_2) \\ f_0(z_1,z_2,z_3) & \longrightarrow & z_1 \text{ if } eq(z_1,z_3) \\ f_0(z_1,z_2,z_3) & \longrightarrow & z_2 \text{ if } eq(z_2,z_3) \end{array}$$

Les parties gauches sont identiques. Le système final est :

$$\begin{array}{rcl} f_0(z_1, z_2, z_3) & \longrightarrow & f_0'(z_1, z_2, z_3, eq(z_1, z_2), eq(z_1, z_3), eq(z_2, z_3)) \\ f_0'(z_1, z_2, z_3, true, b_2, b_3) & \longrightarrow & z_1 \\ f_0'(z_1, z_2, z_3, b_1, true, b_3) & \longrightarrow & z_1 \\ f_0'(z_1, z_2, z_3, b_1, b_2, true) & \longrightarrow & z_2 \end{array}$$

Exemple C.10 *Soit l'opérateur* $f_1 : Nat \times Nat \times Nat \rightarrow Nat$ *défini comme suit :*

$$\begin{array}{rcl} f_1(x, x, y) & \longrightarrow & t_1 \\ f_1(x, succ(y), succ(y)) & \longrightarrow & t_2 \end{array}$$

Cette définition devient :

$$\begin{array}{rcl} f_1(x_1, x_2, y) & \longrightarrow & t_1 \text{ if } x_1 = x_2 \\ f_1(x, succ(y_1), succ(y_2)) & \longrightarrow & t_2 \text{ if } y_1 = y_2 \end{array}$$

Les deux parties gauches sont superposables. Après scission des variables et suppression des règles dont la prémisse est trivialement fausse :

$$\begin{array}{rcll} f_1(0, 0, 0) & \longrightarrow & t_1 & \\ f_1(0, 0, succ(x_5)) & \longrightarrow & t_1 & \\ f_1(succ(x_3), succ(x_4), 0) & \longrightarrow & t_1 & \text{if } x_3 = x_4 \\ f_1(succ(x_3), succ(x_4), succ(x_5)) & \longrightarrow & t_1 & \text{if } x_3 = x_4 \\ f_1(0, succ(x_4), succ(x_5)) & \longrightarrow & t_2 & \text{if } x_4 = x_5 \\ f_1(succ(x_3), succ(x_4), succ(x_5)) & \longrightarrow & t_2 & \text{if } x_4 = x_5 \end{array}$$

Le système final :

$$\begin{array}{rcl} f_1(0, 0, 0) & \longrightarrow & t_1 \\ f_1(0, 0, succ(x_5) & \longrightarrow & t_1 \\ f_1(0, succ(x_4), succ(x_5)) & \longrightarrow & f_1'(0, succ(x_4), succ(x_5), x_4 = x_5) \\ f_1(succ(x_3), succ(x_4), 0) & \longrightarrow & f_1''(succ(x_3), succ(x_4), 0, x_3 = x_4) \\ f_1(succ(x_3), succ(x_4), succ(x_5)) & \longrightarrow & f_1'''(succ(x_3), succ(x_4), succ(x_5), x_3 = x_4, x_4 = x_5) \\ f_1'(0, succ(x_4), succ(x_5), true) & \longrightarrow & t_2 \\ f_1''(succ(x_3), succ(x_4), 0, true) & \longrightarrow & t_1 \\ f_1'''(succ(x_3), succ(x_4), succ(x_5), true, b) & \longrightarrow & t_1 \\ f_1'''(succ(x_3), succ(x_4), succ(x_5), b, true) & \longrightarrow & t_2 \end{array}$$

Nous concluons ce résultat en soulignant que ce sont les cas où les termes ne sont pas réductibles parce que l'opérateur n'est pas complètement défini par rapport aux constructeurs qui justifient cette notion d'"équivalence par rapport aux constructeurs", laquelle notion permet d'introduire sans dommage des opérateurs auxiliaires cachés. Rappelons que cette notion d'"équivalence par rapport aux constructeurs" ne perturbe pas le cadre de notre étude, puisque de tels termes sont réduits à "!*nothing-matches*" après introduction des exceptions. De même, la notion de confluence par rapport aux constructeurs, que nous introduisons au §C.3, à titre de "curiosité exhaustive", se ramène aisément à la définition classique de confluence après introduction de l'exception "!*nothing-matches*".

Concernant les systèmes de réécriture non linéaires à gauche, un résultat analogue est démontré dans [43], mais avec d'autres hypothèses. Hubert Comon prouve que si un opérateur est complètement défini par un système de réécriture confluent, alors, dans le cas de variables dont la sorte est à support fini, on peut scinder les règles non linéaires à gauche (d'après leurs variables répétées), et dans le cas contraire, on peut enlever ces règles. On peut considérer ce résultat comme un cas particulier du corollaire C.8, avec, en sus, l'hypothèse de convergence, et le fait qu'il est possible de *scinder* ou d'*enlever* les règles non linéaires. En fait, le résultat de [43] n'est guère utilisable ici, car nous tenons à relâcher l'hypothèse de définition complète, en ayant à l'esprit la possibilité de la compléter éventuellement au moyen de termes exceptionnels.

C.3 Résultats de confluence

Jusqu'à présent, nous avons considéré des systèmes de réécriture qui n'étaient pas nécessairement confluents, afin de nous placer sous les hypothèses les plus faibles. Nous montrons à présent que si l'on considère une notion de confluence appliquée aux termes formés uniquement de constructeurs, la transformation décrite précédemment préserve cette propriété.

Définition C.11 *Un système de réécriture est dit* **confluent par rapport aux constructeurs** *si et seulement si lorsqu'un terme admet une forme normale pouvant s'exprimer uniquement en fonction de constructeurs, c'est l'*unique *forme normale du terme.*

Proposition C.12 *Soient $\mathcal{R}$ et $\mathcal{S}$ deux systèmes de réécriture équivalents par rapport aux constructeurs. Si $\mathcal{R}$ est confluent par rapport aux constructeurs, alors $\mathcal{S}$ est confluent par rapport aux constructeurs.*

Preuve Soit un terme t_0 dont une forme normale t_1, obtenue au moyen de $\mathcal{R}$, s'exprime uniquement en fonction de constructeurs. D'après les hypothèses, cette forme normale est unique selon $\mathcal{R}$ et atteignable au moyen de $\mathcal{S}$. Soit t_2 une forme normale de t_0 obtenue au moyen de $\mathcal{S}$, et différente de t_1. Examinons les deux cas :

- t_2 peut s'exprimer uniquement en fonction de constructeurs. $\mathcal{R}$ et $\mathcal{S}$ étant équivalents par rapport aux constructeurs, t_2 est atteignable au moyen de $\mathcal{R}$: contradiction. □
- Dans le cas contraire, cela signifierait que t_0 est soit irréductible par $\mathcal{S}$, soit qu'une de ses formes normales ne peut pas s'exprimer uniquement en fonction de constructeurs. $\mathcal{R}$ et $\mathcal{S}$ étant équivalents par rapport aux constructeurs, t_0 vérifie la même propriété pour $\mathcal{R}$: là aussi, contradiction. □

Corollaire C.13 *Pour tout système de réécriture confluent $\mathcal{R}$, opérant sur des termes dont les sortes sont des sortes avec égalité, il existe un système de réécriture équivalent à $\mathcal{R}$ par rapport aux constructeurs, linéaire à gauche, et confluent par rapport aux constructeurs.*

Preuve Ce corollaire est une conséquence directe du théorème C.6 et de la proposition C.12.

Annexe D

Entrées-sorties en FP2

Dans cette annexe, nous allons indiquer comment est réalisée en FP2 l'interaction avec l'utilisateur lors de l'exécution d'un processus, évoquer les réalisations qui ont été nécessaires dans la partie fonctionnelle, et montrer que l'on peut traiter les entrées erronées à l'aide des récupérateurs d'exceptions.

D.1 Processus "clavier" et "écran"

Un langage fonctionnel peut fort bien se passer de primitives d'entrées-sorties : à preuve, ML [89] et MIRANDA [174] n'en proposent pas. On peut en effet considérer que les arguments d'une fonction constituent l'entrée d'un calcul et le résultat sa sortie. Cependant, un autre point est que, en FP2, les seuls processus que l'on peut exécuter sont des processus dont toutes les transitions sont internes, c'est-à-dire qu'il ne doit plus subsister de connecteurs visibles dans ces processus. Ce qui implique de *fermer* un réseau dont on souhaite la simulation. Tous les connecteurs qui sont compris comme des connecteurs d'entrée doivent alors être reliés à des sorties d'un processus clavier ($KEYBOARD$) et ceux qui sont compris comme des connecteurs de sortie à des entrées d'un processus écran ($SCREEN$).

Un processus $SCREEN$ peut être spécifié comme suit :

$$\begin{array}{rcrcl} & & & ==> & INIT \\ INIT & : & PRINT_0(x) & ==> & A(print(x)) \\ A(x) & : & & ==> & INIT \end{array}$$

en convenant que la fonction *print*, qui écrit son argument sur l'écran, est fonctionnellement équivalente à l'identité, c'est-à-dire qu'elle retourne son argument. Dans le contexte de FP2, on voit ici l'utilité d'une fonction *print* comme celle d'une "fonction" *read* pour spécifier un processus $KEYBOARD$:

$$\begin{array}{rcl} & ==> & A(read) \\ A(x) : READ_0(x) & ==> & A(read) \end{array}$$

où *read* retourne la forme normale d'un terme lu.

Remarque En fait, ce processus $KEYBOARD$ n'est pas idéal : il est en avance d'une lecture par rapport aux envois via $READ_0$. Précisons également que d'une façon générale, la saisie des entrées ne doit pas être bloquante, car sinon, cela signifie que si le réseau choisit d'exécuter une règle avec lecture dans la postcondition, cette dernière *doit* alors être satisfaite. Il est donc nécessaire de pouvoir tester la vacuité du flux d'entrée afin, le cas échéant, de ne pas exécuter une telle règle. Nous n'abordons pas davantage ces problèmes et, pour de

plus amples détails sur ces sujets, nous renvoyons le lecteur à [161, 162] dans le cadre de μFP2, et... au futur manuel d'utilisation de FP2.

Un problème se pose à présent : il serait souhaitable que ces processus soient utilisables quelque soit la sorte des connecteurs, c'est-à-dire qu'il soient paramétrés par la propriété *Ftype*, ce qui imposerait la même paramétrisation pour les opérateurs *read* et *print*. Si donc nous considérons l'opérateur *print* comme un opérateur générique, cela oblige que nous soyons capable, étant donné un terme Lisp issu de la compilation d'un terme FP2 quelconque, de retrouver directement sa représentation externe. Autrement dit, soit plusieurs sortes ne peuvent pas partager la même représentation, soit il faut renoncer au fait que l'on ignore durant une exécution la sorte des objets que l'on manipule — c'est le côté "fortement typé" du langage — et conserver tout au long des évaluations une information de sorte. Le même inconvénient existe pour les exceptions avec paramètres génériques, comme nous l'avons vu au §4.2.4, et, bien sûr, avec l'opérateur *read*, si l'on veut pouvoir vérifier que le terme que l'on va lire est bien de la sorte attendue.

En fait, ces opérateurs d'entrées-sorties sont des "perversions" qui ont été introduites dans le langage à seule fin de spécifier les processus clavier et écran. Étant donné que ces processus sont ajoutés "au dernier moment" pour fermer un réseau que l'on désire simuler — on peut supposer par conséquent que ce dernier est complètement instancié —, la solution retenue a donc été de ne permettre l'utilisation des "opérateurs" d'entrées-sorties que sur des termes complètement instanciés. Il est alors possible de prévoir à la compilation une fonction d'écriture ou de lecture *ad hoc* pour chaque sorte utilisée. Il s'ensuit qu'il existe également autant de processus clavier et écran que d'utilisations avec des sortes différentes[86].

D.2 Réalisation — Exemples

Nous avons vu (cf. §2.1.1) que la fonction d'écriture de la boucle d'évaluation de FP2 admet pour arguments un terme et sa sorte. Pour chaque sorte *sort-expr*, *complètement instanciée* (la condition de fermeture du chapitre 3 ne suffit pas), il est possible de définir une fonction Lisp `print-sort-expr` comme suit :

```
(defun print-sort-expr (x)
  (fp2-print x 'sort-expr))
```

où `fp2-print` est la fonction Lisp qui réalise la conversion d'un terme compilé en sa forme externe, et l'affichage de cette dernière.

De même, il est possible d'écrire en Lisp une fonction "générique" de lecture admettant un argument sorte et de définir de façon analogue `read-sort-expr`. Par conséquent, à la différence des autres opérateurs dont l'accès au nom interne s'effectue d'après le profil, c'est le codomaine qui est la clé d'accès aux divers exemplaires internes des opérateurs *read* et *print*. Si la sorte ne figure pas dans les accès possibles, la fonction Lisp correspondante est définie et une nouvelle clé est ajoutée.

[86] Dans le but de diminuer le nombre des règles de transition, on peut alors écrire des processus écran à une seule règle de transition, avec un paramètre d'initialisation indifférent. Par exemple, pour *SCREEN-NAT* :

$$\begin{array}{rcl} & ==> & A(0) \\ A(x) : PRINT\text{-}NAT_0(y) & ==> & A(print(y)) \end{array}$$

Au niveau de l'analyse des sortes, tout se passe comme si les profils respectifs des opérateurs *read* et *print* étaient "$- \rightarrow s$" et "$s \rightarrow s$", où s est une variable de travail à substituer. Du fait de son profil, l'opérateur *read* est assujetti aux contraintes des opérateurs à paramètre générique présent dans le codomaine et absent dans le domaine. Ainsi, le terme "*read*" seul est ambigu, tandis que dans "$0 + read$", "*read*" désigne "$read : Nat$". À noter que la sorte attendue d'une lecture est utilisée lors de l'analyse sémantique de cette dernière (voir l'exemple suivant).

Exemple D.1 *Session avec entrées-sorties. (Nous supposons qu'aucune fonction de lecture ou d'écriture n'a été définie durant les évaluations qui ont précédé cette session. Les mentions 'définition de "read-..."' et 'définition de print-..."' signifient que le compilateur génère de nouvelles fonctions Lisp pour réaliser les entrées-sorties.)*

$read : Seq[Nat]$ \$ -- ("\$" marque la fin d'une expression.)
-- définition de "$read\text{-}Seq[Nat]$".
nil \$ -- analysé comme "$nil : Seq[Nat]$".
nil

$read : Nat$ \$
-- définition de "$read\text{-}Nat$".
$2 + length([[true]] + read)$ \$ -- "*read*" compris comme "$read : Seq[Seq[Bool]]$.
-- définition de "$read\text{-}Seq[Seq[Bool]]$".
$[\overbrace{read}^{(1)}] + print(\overbrace{read}^{(2)}) + [[nil?(\overbrace{read}^{(3)} : Seq[Nat])]]$ \$
-- définition de "$read\text{-}Seq[Bool]$" et de "$print\text{-}Seq[Seq[Bool]]$" ;
-- "$read\text{-}Seq[Seq[Bool]]$" et "$read\text{-}Seq[Nat]$" déjà définis.
-- (Il est bien sûr nécessaire de convenir d'un ordre d'évaluation :
-- en l'occurrence, de gauche à droite.)
$[false]$ \$ -- évaluation de (1).
nil \$ -- évaluation de (2).
nil -- effet de "*print*".
$head(\overbrace{read}^{(4)})$ \$ -- évaluation de (3).
$[[0, 1], [2]]$ \$ -- évaluation de (4).
5 -- résultat (ouf !).

$read : Nat + read$ \$ -- Certaines lectures peuvent ne pas être effectuées en cas de
$1\ div\ 0$ \$ -- déclenchement d'exception.
!zero-divide

D.3 Récupération d'entrées erronées

Si nous convenons qu'en cas d'échec des analyses syntaxique puis sémantique d'une expression, le résultat de l'évaluation de ce terme est l'exception "*!wrong-expr*", il est dès lors possible de programmer un opérateur de lecture qui, à chaque fois que l'entrée est mal formée, demande une entrée de remplacement.

Exemple D.2 *Remplacement d'entrées entières erronées pour cause d'échec des analyses syntaxique ou sémantique.*

```
enr
    opns  super-read-Nat  :  –  →  Nat
    rules
        <>  super-read-Nat  ==>  rec  read
                                      when !wrong-expr then super-read-Nat
                                 endrec
endenr
```

zzero $
SEMANTIC ERROR: operator *zzero* is unknown
SEMANTIC ERROR: no possible semantic analysis
!*wrong-expr*

super-read-Nat $
zzero $
SEMANTIC ERROR: operator *zzero* is unknown
SEMANTIC ERROR: no possible semantic analysis
zero $
0

Cet opérateur *super-read-Nat* est donc capable de récupérer les entrées mal formées, mais *pas* les entrées dont l'évaluation déclenche une exception.

Exemple D.3 *Remplacement de toutes les entrées entières qui déclenchent une exception.*

```
enr
    opns  super-super-read-Nat  :  –  →  Nat
    rules
        <>  super-super-read-Nat  ==>  rec  read
                                            otherwise super-super-read-Nat
                                       endrec
endenr
```

super-super-read-Nat $
1 *div* 0 $
1 *div true* $
SEMANTIC ERROR: no possible semantic analysis
0 *div* 1 $
0

Bibliographie

[1] ARKAXHIU (Eqerem) : *Un environnement et un langage graphique pour la spécification de processus parallèles communicants.* Thèse de 3ème cycle. Grenoble, juillet 1984.

[2] AUSTRY (Didier) : *Aspects syntaxiques de Meije, un calcul pour le parallélisme. Applications.* Thèse de 3ème cycle. Paris VII, 1982.

[3] BACHMAIR (Leo), DERSHOWITZ (Nachum) : *Commutation, Transformation and Termination.* Proc. CADE–8. Oxford, 1986. LNCS no. 230, pp. 5–20. Springer-Verlag.

[4] BACKUS (John W.) : *Can Programming Be Liberated From The Von Neumann Style? A Functional Style and Its Algebra of Programs.* CACM. Vol. 21, pp. 613–641. August 1978.

[5] BAETEN (J.C.M.), BERGSTRA (J.A.), KLOP (J.W.) : *Term Rewriting Systems with Priorities.* Proc. RTA'87. Bordeaux, May 1987. LNCS no. 256, pp. 83–94. Springer-Verlag.

[6] BARENDREGT (Hendrik Pieter) : *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, Vol. 103. Revised Edition, 1984. North-Holland.

[7] BELLOT (Patrick) : *Sur les sentiers du GRAAL, étude, conception et réalisation d'un langage de programmation sans variable.* Thèse d'État. LITP 86–62. Paris VII, octobre 1986.

[8] BELMESK (Zoubir) : *Méthodologie de spécification et de programmation des protocoles de communication avec le langage FP2. Cas étudié : le niveau liaison du protocole X25.* R.R. 748–I–IMAG 86 LIFIA. Grenoble, octobre 1988.

[9] BENSALEM (Saddek) : *Algèbre de programmes dans un univers typé.* Thèse de 3ème cycle. Grenoble, décembre 1985.

[10] BERNOT (Gilles) : *Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs ; application à l'implémentation et aux primitives de structuration des spécifications formelles.* Thèse de 3ème cycle. Orsay, février 1986.

[11] BERNOT (Gilles) : *Good Functors ... Are Those Preserving Philosophy!* Proc. of Category Theory & Computer Science. Edinburgh, September 1987. LNCS no. 283, pp. 182–195. Springer-Verlag.

[12] BERNOT (Gilles), BIDOIT (Michel), CHOPPY (Christine) : *Algebraic Semantics of Exception Handling.* Proc. ESOP'86. Saarbrücken, March 1986. LNCS no. 213, pp. 173–186. Springer-Verlag.

[13] BERNOT (Gilles), BIDOIT (Michel), CHOPPY (Christine) : *Abstract Data Types with Exception Handling: an Initial Approach Based on a Distinction between Exceptions and Errors.* TCS 46, pp. 13–45. Elsevier Science Publishers, B.V. (North-Holland), 1986.

[14] BERRY (Gérard), COURONNÉ (Philippe), GONTHIER (Georges) : *Programmation synchrone des systèmes réactifs : le langage ESTEREL.* TSI. Vol. 6, n° 4, pp. 305–316. Dunod, juillet-août 1987.

[15] BERT (Didier) : *La programmation générique. Construction de logiciel, spécification algébrique et vérification.* Thèse d'État. Grenoble, juin 1979.

[16] BERT (Didier) : *Refinements of Generic Specifications with Algebraic Tools.* Proc. IFIP'83, R.E.A. Mason (ed.), pp. 815–820. Elsevier Science Publishers B.V. (North-Holland). Paris, September 1983.

[17] BERT (Didier) : *Manuel de référence de LPG, Version 1.2.* RR IMAG 408. Grenoble, décembre 1983.

[18] BERT (Didier), DRABIK (Pascal) : *LPG : structuration des spécifications et validation sémantique autorisée.* RR 648-I-IMAG-58 LIFIA. Grenoble, mars 1987.

[19] BERT (Didier), DRABIK (Pascal), ECHAHED (Rachid) : *Manuel de référence de LPG version 1.8.* RT 17-IMAG-1 LIFIA. Grenoble, mars 1987.

[20] BERT (Didier), ECHAHED (Rachid) : *Design and Implementation of a Generic, Logic and Functional Programming Language.* Proc. ESOP'86. Saarbrücken, March 1986. LNCS no. 213, pp. 119–132. Springer-Verlag.

[21] BIBEL (Wolfgang) : *Matings in Matrices.* CACM. Vol. 26, no. 11, pp. 844–852. November 1983.

[22] BIDOIT (Michel) : *Une méthode de présentation des types abstraits : applications.* Thèse de 3ème cycle. Orsay, juin 1981.

[23] BIDOIT (Michel), CAPY (Francis), CHOPPY (Christine), CHOQUET (Nicole), GRESSE (Christian), KAPLAN (Stéphane), SCHLIENGER (Françoise), VOISIN (Fédéric) : *ASSPRO : un environnement de programmation interactif et intégré.* TSI. Vol. 6, n° 1, pp. 21–40. Dunod, janvier-février 1987.

[24] BIDOIT (Michel), CHOPPY (Christine), VOISIN (Frédéric) : *The ASSPEGIQUE Specification Environment: Motivations and Design.* Recent Trends in Data Type Specification. Informatik-Fachberichte 116. Springer-Verlag, 1985.

[25] BIRKHOFF (Garrett) : *On the Structure of Abstract Algebras.* Proc. of the Cambridge Philosophical Society, 31, pp. 433–454. 1935.

[26] BOUSDIRA (Wadoud) : *A Completion Procedure for Hierarchical Conditional Rewriting Systems.* Proc. of an International Workshop. Gaussig (GDR), November 1988. In: "Algebraic and Logic Programming", pp. 93–107. Band 49. Akademie-Verlag. Berlin, GDR.

[27] BOUSDIRA (Wadoud), RÉMY (Jean-Luc) : *Hierarchical Contextual Rewriting with Several Levels.* Proc. STACS'88. Bordeaux, February 1988. LNCS no. 294, pp. 193–206. Springer-Verlag.

[28] BOYER (Robert S.), MOORE (J. Strother) : *A Fast String Searching Algorithm.* CACM. Vol. 20, no. 10, pp. 762–772. October 1977.

[29] BRETZ (Manfred), EBERT (Jürgen) : *An Exception Handling Construct for Functional Languages.* Proc. ESOP'88. Nancy, March 1988. LNCS no. 300, pp. 160–174. Springer-Verlag.

[30] BROY (Manfred), WIRSING (Martin) : *Partial Abstract Types.* Acta Informatica. Vol. 18, pp. 47–64. 1982.

[31] BROY (Manfred), WIRSING (Martin) : *Algebraic Definition of a Functional Programming Language and Its Semantic Models.* RAIRO Informatique Théorique. Vol. 17, no. 2, pp. 137–161. 1983.

[32] BURSTALL (Rod M.), DARLINGTON (John) : *A Transformation System for Developing Recursive Programs.* JACM. Vol. 24, no. 1, pp. 44–67. January 1977.

[33] BURSTALL (Rod M.), GOGUEN (Joseph A.) : *Putting Theories Together to Make Specifications.* Proc. of the 5th International Joint Conference on Artificial Intelligence, pp. 1045–1058. Cambridge (Massachusetts), 1977.

[34] BURSTALL (Rod M.), GOGUEN (Joseph A.) : *The Semantics of CLEAR, a Specification Language.* Proc. 1979 Copenhagen Winter School on Abstract Software Specifications. LNCS no. 86, pp. 292–332.

[35] BURSTALL (Rod M.), MACQUEEN (D.B.), SANNELLA (Donald T.) : *HOPE: An Experimental Applicative Language.* Internal Report CSR-62-80. Edinburgh, February 1981.

[36] BUSH (V.J.), GURD (J.R.) : *Transforming Recursive Programs for Execution on Parallel Machines.* Proc. Functional Programming Languages and Computer Architecture. Nancy, September 1985. LNCS no. 201, pp. 350–367. Springer-Verlag.

[37] CAFERRA (Ricardo), JORRAND (Philippe) : *Unification in Parallel with Refined Linearity Test: an Example of Recursive Network Structure in FP2, a Functional Parallel Programming Language.* Proc. EUROCAL'85. Vol. 2: Research Contributions. Linz, April 1985. LNCS no. 205, pp. 539–540. Springer-Verlag.

[38] CASPI (Paul), HALBWACHS (Nicolas), PILAUD (Daniel), PLAICE (John Alexander) : *LUSTRE: a Declarative Language for Programming Synchronous Systems.* Proc. 14th ACM POPL, pp. 178–188. Munich, 1987.

[39] CHURCH (Alonzo) : *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[40] CISNEROS GASCON (Maria del Pilar) : *Programmation parallèle et programmation fonctionnelle : propositions pour un langage.* Thèse de 3ème cycle. Grenoble, octobre 1984.

[41] CLOCKSIN (William F.), MELLISH (Christopher S.) : *Programming in Prolog.* Springer-Verlag, 1984.

[42] COLMERAUER (Alain), KANOUI (Henry), VAN CANEGHEM (Michel) : *Prolog, bases théoriques et développements actuels.* TSI. Vol. 2, n° 4, pp. 271–311. Dunod, juillet-août 1983.

[43] COMON (Hubert) : *Unification et disunification. Théorie et applications.* Thèse de l'INPG. Grenoble, mars 1988.

[44] CORBIN (Jacques), BIDOIT (Michel) : *A Rehabilitation of Robinson's Unification Algorithm.* Proc. IFIP'83, R.E.A. Mason (ed.), pp. 909–914. Elsevier Science Publishers B.V. (North-Holland). Paris, September 1983.

[45] COUSINEAU (Guy), CURRIEN (Pierre-Louis), MAUNY (Michel) : *The Categorical Abstract Machine.* Proc. Functional and Programming Languages and Computer Architecture. Nancy, September 1985. LNCS no. 201, pp. 50–64. Springer-Verlag.

[46] CRISTIAN (Flaviu) : *Exception Handling and Software Fault Tolerenace.* IEEE Trans. on Computer Science, pp. 531–540. June 1982.

[47] CRISTIAN (Flaviu) : *Robust Data Types.* Acta Informatica. Vol. 17, pp. 365–397. Springer-Verlag, 1982.

[48] CRISTIAN (Flaviu) : *Reasonning about Programs with Exceptions.* Proc. 30th International Symposium on Fault-Tolerant Computing. IEEE pp. 188–195. Milano, June 1983.

[49] CRISTIAN (Flaviu) : *Exceptions, défaillances et erreurs.* TSI. Vol. 4, n° 4, pp. 385–390. Dunod, juillet-août 1985.

[50] DARLINGTON (John) : *Program Transformation.* In: "Functional Programming and Its Applications. An Advanced Course", pp. 193–215. Cambridge University Press, 1982.

[51] DERSHOWITZ (Nachum) : *Termination of Rewriting.* Journal of Symbolic Computation (1987) 3, pp. 69–116. April 1987.

[52] DERSHOWITZ (Nachum), JOUANNAUD (Jean-Pierre) : *Rewrite Systems.* "Handbook of Theoretical Computer Science", Chapter 15. North-Holland, 1989.

[53] DRABIK (Pascal) : *Exemple de validation sémantique dans les théories structurées.* LIFIA. RT 45. Grenoble, octobre 1988.

[54] DWORK (Cynthia), KANELLAKIS (Paris C.), MITCHELL (John C.) : *On the Sequential Nature of Unification.* Journal of Logic Programming 1984: 1, pp. 35–50. Elsevier Science Publishing Co, Inc. New York.

[55] ECHAHED (Rachid) : *Prédicats et sous-types en LPG. Réalisation de la E-unification.* RR IMAG 550 (LIFIA 29). Grenoble, juillet 1985.

[56] EHRIG (H-D.) : *On the Theory of Specification, Implementation, and Parametrization of Abstract Data Types.* JACM. Vol. 29, no. 1, pp. 206–227. January 1982.

[57] EHRIG (Hartmut), KREOWSKI (Hans-Jörg) : *Compatibility of Parameter Passing and Implementation of Parameterized Data Types.* TCS 27, pp 255–286. Elsevier Science Publishers B.V. (North-Holland), 1983.

[58] EHRIG (Hartmut), KREOWSKI (Hans-Jörg), THATCHER (James W.), WAGNER (Eric G.) : *Parameter Passing in Algebraic Specification Languages.* TCS 28, pp. 45–81. Elsevier-Science Publishers B.V. (North-Holland), 1984.

[59] EHRIG (Hartmut), MAHR (Bernd) : *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer-Verlag, 1985.

[60] FAGES (François) : *Formes canoniques dans les algèbres booléennes, et application à la démonstration automatique en logique de premier ordre.* Thèse de 3ème cycle. Paris VI, juin 1983.

[61] FRADET (Pascal) : *Compilation des langages fonctionnels par transformation de programmes.* Thèse de l'Université de Rennes I. Novembre 1988.

[62] FRADET (Pascal), LE MÉTAYER (Daniel) : *Compilation of Lambda-Calculus into Functional Machine Code.* Proc. TAPSOFT'89. Vol. II. LNCS no. 352, pp. 155–166. Springer-Verlag.

[63] FUTATSUGI (Kokichi), GOGUEN (Joseph A.), JOUANNAUD (Jean-Pierre), MESEGUER (José) : *Principles of OBJ2.* Proc. 12th ACM POPL, pp. 52–66. New Orleans (Louisiana), January 14–16, 1985.

[64] FUTATSUGI (Kokichi), GOGUEN (Joseph A.), MESEGUER (José), OKADA (Koji) : *Parameterized Programming in OBJ2.* Proc. 9th International Conference on Software Engineering. Monterey (California). March 30–April 2, 1987. ACM, pp. 51–60.

[65] GABRIEL (Peter), ULMER (Friedrich) : *Lokal präsentierbare Kategorien.* Lecture Notes in Mathematics, no. 221. Springer-Verlag.

[66] GAMATIE (Boubakar) : *Vérification statique de traitement d'exceptions dans les programmes parallèles. Conception et mise en œuvre de mécanismes adaptés.* Thèse de 3ème cycle. Rennes, avril 1982.

[67] GARAVEL (Hubert) : *Compilation et vérification de programmes LOTOS.* Thèse de l'INPG, en préparation. 1989.

[68] GAUTIER (Thierry), LE GUERNIC (Paul) : *SIGNAL: a Declarative Language for Synchronous Programming of Real-Time Systems.* Proc. FPLCA'87. Portland (Oregon), September 1987. LNCS no. 274, pp. 257–277. Springer-Verlag.

[69] GOGOLLA (Martin) : *Über partiell geordnete Sortenmengen und deren Anwendung zur Fehlerbehandlung in abstrakten Datentypen.* Dissertation. Technische Universität Carolo-Wilhelmina zu Braunschweig. Juni 1986.

[70] GOGOLLA (Martin), DROSTEN (K.), LIPECK (U.), EHRIG (H.-D.) : *Algebraic and Operational Semantics of Specifications Allowing Exceptions and Errors.* TCS 34, pp. 289–313. Elsevier-Science Publishers B.V. (North-Holland), 1984.

[71] GOGUEN (Joseph A.) : *Abstract Errors for Abstract Data Types.* Proc. IFIP Working Conference on Formal Description of Programming Concepts. MIT, 1977.

[72] GOGUEN (Joseph A.) : *Parameterized Programming.* IEEE Transactions on Software Engineering. Vol. SE–10, no. 5, pp. 528–543. September 1984.

[73] GOGUEN (Joseph A.) : *Higher Order Functions Considered Unnecessary for Higher Order Programming.* SRI–CSL–88–1R. SRI Projects 1243, 2316, and 4415. April 1988.

[74] GOGUEN (Joseph A.) : *OBJ as a Theorem Prover with Applications to Hardware Verification.* SRI–CSL–88–4R2. SRI Projects 1243, 2316 and 4415. August 1988.

[75] GOGUEN (Joseph A.) : *Modular Algebraic Specification of Some Basic Geometrical Constructions.* Artificial Intelligence. Vol. 37, nos. 1–3, pp. 123–153. North-Holland, December 1988.

[76] GOGUEN (Joseph A.), BURSTALL (Rod M.) : *Some Fundamentals Algebraic Tools for the Semantic of Computation. Part 1: Comma Categories, Colimits, Signatures and Theories.* TCS 31, pp. 175–209. Elsevier Science Publishers, B.V. (North-Holland), 1984.

[77] GOGUEN (Joseph A.), BURSTALL (Rod M.) : *Some Fundamentals Algebraic Tools for the Semantic of Computation. Part 2: Signed and Abstract Theories.* TCS 31, pp. 263–295. Elsevier Science Publishers, B.V. (North-Holland), 1984.

[78] GOGUEN (Joseph A.), BURSTALL (Rod M.) : *Institutions: Abstract Model Theory for Computer Science.* Draft, February 1986.

[79] GOGUEN (Joseph A.), JOUANNAUD (Jean-Pierre), MESEGUER (José) : *Operational Semantics for Order-Sorted Algebra.* CRIN 84–R–101. Nancy, 1984.

[80] GOGUEN (Joseph A.), KIRCHNER (Claude), KIRCHNER (Hélène), MÉGRELIS (Aristide), MESEGUER (José), WINKLER (Thimothy C.) : *An Introduction to OBJ3.* CRIN 88–R–001. Nancy, 1988.

[81] GOGUEN (Joseph A.), KIRCHNER (Claude), MESEGUER (José), WINKLER (Thimothy C.) : *OBJ as a Language for Concurrent Programming.* Proc. Second International Conference on Supercomputing. Santa Clara, May 1987.

[82] GOGUEN (Joseph A.), MESEGUER (José) : *Programming with Parameterized Abstract Objects in OBJ.* In: "Theory and Practice of Software Technology". North-Holland, 1983.

[83] GOGUEN (Joseph A.), MESEGUER (José) : *EQLOG: Equality, Types, and Generic Modules for Logic Programming.* In: "Functional and Logic Programming". Prentice-Hall, 1985.

[84] GOGUEN (Joseph A.), MESEGUER (José) : *Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems.* Proc. Symposium on Logic in Computer Science. Ithaca, New York, June 1987. ACM, pp. 18–29.

[85] GOGUEN (Joseph A.), MESEGUER (José) : *Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism, and Partial Operations.* Draft. SRI International, May 1988. (Previous version given as lecture "Logic and Subsorts and Polymorphism" at Seminar on Types, Carnegie-Mellon University, June 1983.)

[86] GOGUEN (Joseph A.), THATCHER (James W.), WAGNER (Eric G.) : *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* Current Trends in Programming Methodology. Vol. 4: Data Structuring, chap. 5. Prentice-Hall, 1978.

[87] GOGUEN (Joseph A.), THATCHER (James W.), WAGNER (Eric G.), WRIGHT (J.B.) : *Initial Algebra Semantics and Continuous Algebras.* JACM. Vol. 24, no. 1, pp. 68–95. January 1977.

[88] GOGUEN (Joseph A.), WINKLER (Timothy C.) : *Introducing OBJ3.* SRI–CSL–88–9. SRI Projects 1243, 2316, and 4415. August 1988.

[89] GORDON (Michael J.), MILNER (Arthur J.), WADSWORTH (Christopher P.) : *Edinburgh LCF.* LNCS no. 78. Springer-Verlag, 1979.

[90] GRÄTZER (George) : *Universal Algebra.* Second Edition, 1979. Springer-Verlag.

[91] GUTTAG (John V.), HORNING (James J.) : *The Algebraic Specification of Abstract Data Types.* Acta Informatica. Vol. 10, pp. 27–52. Springer-Verlag, 1978.

[92] GUTTAG (John V.), HORNING (James J.) : *Preliminary Report on The Larch Shared Language.* MIT, October 1983.

[93] GUTTAG (John V.), HOROWITZ (Ellis), MUSSER (David R.) : *Abstract Data Types and Software Validation.* CACM. Vol. 21, no. 12, pp. 1048–1063. December 1978.

[94] HENDERSON (P.), MORRIS (J.) : *A Lazy Evaluator.* Proc. 3rd ACM POPL, pp. 95–103. Atlanta (Georgia), 1976.

[95] HEUILLARD (Thierry) : *Compiling Conditional Rewriting Systems.* Proc. 1st International Workshop on Conditional Term Rewriting Systems. Orsay, July 1987. LNCS no. 308, pp. 111–128. Springer-Verlag.

[96] HOARE (Charles Anthony Richard) : *Communicating Sequential Processes.* CACM. Vol. 21, no. 8, pp. 666–677. August 1978.

[97] HUET (Gérard) : *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω.* Thèse d'État. Paris VII, septembre 1976.

[98] HUET (Gérard) : *Confluent Reductions : Abstract Properties and Applications to Term Rewriting Systems.* JACM. Vol. 27, no. 4, pp. 797–821. October 1981.

[99] HUET (Gérard) : *Initiation à la Théorie des Catégories.* Notes de cours, 1987.

[100] HUET (Gérard) : *A Uniform Approach to Type Theory.* INRIA, RR no. 795. February 1988.

[101] HUET (Gérard), LÉVY (Jean-Jacques) : *Computations in Nonambiguous Linear Rewriting Systems.* INRIA, August 1979.

[102] HUET (Gérard), OPPEN (Derek C.) : *Equations and Rewrite Rules. A Survey.* In: "Formal Language Theory. Perspectives and Open Problems", pp. 309–405. Ed. R. Book. Academic Press. January 1980.

[103] HUFFLEN (Jean-Michel) : *Notes sur FP et son implantation en LPG.* RR IMAG 518 (LIFIA 20). Grenoble, mars 1985.

[104] HUFFLEN (Jean-Michel) : *Un exemple d'utilisation de FP2 : description d'une architecture parallèle et pipe-line pour l'unification.* RR IMAG 575 (LIFIA 43). Grenoble, janvier 1986.

[105] HUFFLEN (Jean-Michel) : *Une implémentation de LPG en Lisp : "Grand-Guignol".* RR 654–I–IMAG–60 LIFIA. Grenoble, mars 1987.

[106] HUFFLEN (Jean-Michel) : *Parallelizing Recursive Programs.* Proc. AIMSA'88. Varna (Bulgaria), September 1988. North-Holland Publishers, pp. 139–148. (Extended version: R.R. 728–I–IMAG–83 LIFIA. Grenoble, May 1988.)

[107] IBÁÑEZ-ESPIGA (María Blanca) : *Parallel Inferencing in First Order Logic.* Proc. AIMSA'88. Varna (Bulgaria), September 1988. North-Holland Publishers, pp. 149–157.

[108] JACQUET (Paul) : *Les types génériques. Propositions pour un mécanisme d'abstraction dans les langages de programmation.* Thèse de 3ème cycle. Grenoble, septembre 1978.

[109] JORRAND (Philippe) : *Description and Composition of Communicating Processes. Problems of Analysis and Correctness.* R.R. IMAG 290. Grenoble, February 1982.

[110] JORRAND (Philippe) : *FP2: Functional Parallel Programming Based on Term Substitution.* Proc. AIMSA'84. Varna (Bulgaria), September 1984. North-Holland Publishers, pp. 95–112.

[111] JORRAND (Philippe) : *Term Rewriting as a Basis for the Design of a Functional and Parallel Programming Language. A Case of Study: the Language FP2.* In: "Fundamentals of Artificial Intelligence. An Advanced Course". LNCS no. 232, pp. 221–276. Springer-Verlag, 1985.

[112] JORRAND (Philippe) : *Parallélisation de fonctions récursives en FP2.* Notes de séminaire, 1986.

[113] JORRAND (Philippe) : *Design and Implementation of a Parallel Inference Machine for First Order Logic: an Overview.* Proc. PARLE'87. Vol. I: Parallel Architectures. Eindhoven, June 1987. LNCS no. 258, pp. 434–445. Springer-Verlag.

[114] JORRAND (Philippe), HUFFLEN (Jean-Michel), MARTY (Annick), MARTY (Jean-Charles), SCHNOEBELEN (Philippe) : *FP2. The Language and Its Formal Definition.* R.R. IMAG 537 (LIFIA 26). Grenoble, May 1985. (Informal introduction to FP2 available as [111].)

[115] JORRAND (Philippe), PEREIRA-FERNANDEZ (Juan Manuel) : *A Formal Language for Specification of Communicating Processes.* R.R. IMAG 527 (LIFIA 25). Grenoble, 1985.

[116] JOUANNAUD (Jean-Pierre), KIRCHNER (Hélène) : *Completion of a Set of Rules Modulo a Set of Equations.* SIAM Journal of Computing 15 (1), 1986. (Preliminary version in Proc. 11th ACM POPL. Salt Lake City, 1984.)

[117] JOUANNAUD (Jean-Pierre), KIRCHNER (Claude), KIRCHNER (Hélène), MÉGRELIS (Aristide) : *OBJ: Programming with Equalities, Subsorts, Overloading and Parameterization.* Proc. of an International Workshop. Gaussig (GDR), November 1988. In: "Algebraic and Logic Programming", pp. 41–52. Band 49. Akademie-Verlag. Berlin, GDR..

[118] JOUANNAUD (Jean-Pierre), LESCANNE (Pierre) : *La réécriture.* TSI. Vol. 5, n° 6, pp. 433–452. Dunod, novembre-décembre 1986.

[119] KAMIN (S.), LÉVY (Jean-Jacques) : *Two Generalizations of the Recursive Path Ordering.* Unpublished note. Department of Computer Science, University of Illinois. Urbana, 1980.

[120] KAPLAN (Stéphane) : *Conditional Rewrite Rules.* TCS 33, pp. 175–193. Elsevier Science Publishers, B.V. (North-Holland), 1984.

[121] KAPLAN (Stéphane) : *A Compiler for Conditional Term Rewriting Systems.* Proc. RTA'87. Bordeaux, May 1987. LNCS no. 256, pp. 25–41. Springer-Verlag.

[122] KAPLAN (Stéphane), RÉMY (Jean-Luc) : *Completion Algorithms for Conditional Rewriting Systems.* Proc. Colloquium on the Resolution of Equations in Algebraic Structures. Austin, 1987.

[123] KELLER (R.M.) : *Formal Verification of Parallel Programs.* CACM. Vol. 19, no. 7, pp. 371–384. July 1976.

[124] KIRCHNER (Claude) : *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles.* Thèse d'État. Nancy, juin 1985.

[125] KIRCHNER (Claude), KIRCHNER (Hélène), MÉGRELIS (Aristide) : *OBJ for OBJ.* CRIN 87–R–085. Nancy, 1987.

[126] KIRCHNER (Claude), KIRCHNER (Hélène), MESEGUER (José) : *Operational Semantics of OBJ3.* CRIN 87–R–087. Nancy, 1987. (Extended Abstract in: Proc. Automata, Languages and Programming. Tampere, July 1988. LNCS no. 317, pp. 287–301. Springer-Verlag.)

[127] KNUTH (Donald E.), MORRIS JR (James H.), PRATT (Vaughan R.) : *Fast Pattern Matching in Strings.* SIAM Journal of Computing. Vol. 6, no. 2, pp. 323–350. June 1977.

[128] KURFEß (Franz J.), PANDOLFI (Xavier), BELMESK (Zoubir), ERTEL (Wolfgang), LETZ (R.), SCHUMANN (Johannes) : *PARTHEO and FP2: Design of a Parallel Inference Machine.* In: "Parallel Computers: Object-Oriented, Functional, Logic". Wiley, to appear.

[129] LASSEZ (Jean-Louis), MAHER (Michael J.), MARRIOT (Kimbal G.) : *Unification Revisited.* In: "Foundations of Logic and Functional Programming". Workshop. Trento (Italy), December 1986. LNCS no. 306, pp. 67–113. Springer-Verlag.

[130] LE CERTEN (Pascale) : *Conception et mise en œuvre d'un langage impératif pour la programmation parallèle.* Thèse de l'Université de Rennes I, avril 1986.

[131] LEIVANT (Daniel) : *Polymorphic Type Inference.* Proc. 10th ACM POPL, pp. 88–98. Austin (Texas), 1983.

[132] LISKOV (Barbara), GUTTAG (John V.) : *Abstraction and Specification in Program Development.* MIT Press, 1986.

[133] MACCARTHY (John) : *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* CACM. Vol. 3, no. 4, pp. 184–195. April 1960.

[134] MAC LANE (Saunders) : *Categories for the Working Mathematician.* Graduate Texts in Mathematics, Vol. 5. Springer-Verlag, 1971.

[135] MARTELLI (Alberto), MONTANARI (Ugo) : *An Efficient Unification Algorithm.* ACM TOPLAS. Vol. 4, no. 2, pp. 258–282. April 1982.

[136] MARTY (Annick) : *Placement d'un réseau de processus communicants décrit en FP2 sur une structure de grille en vue d'une implantation parallèle de ce langage.* RR IMAG 606 LIFIA 49 I. Grenoble, mai 1986.

[137] MAY (David) : *OCCAM.* Sigplan Notices. Vol. 18, no. 4. ACM, April 1983.

[138] MÉGRELIS (Aristide) : *A Logic of Semi-Functions, Inclusion and Equality. The Setting.* CRIN 89-R-059. June 1989.

[139] MILNER (Robin) : *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences, 17. 1978.

[140] MILNER (Robin) : *A Calculus for Communicating Systems.* LNCS no. 92. Springer-Verlag, 1980.

[141] MILNER (Robin) : *Calculi for Synchrony and Asynchrony.* TCS 25, pp. 267–310. Elsevier Science Publishers, B.V. (North-Holland), 1983.

[142] PADAWITZ (Peter) : *Computing in Horn Clause Theories.* EATCS Monographs on Theoretical Computer Science, Vol. 16. Springer-Verlag, 1988.

[143] PANDOLFI (Xavier) : *Étude de l'inférence de types en FP2, langage fonctionnel et parallèle fondé sur la réécriture de termes.* Rapport de DEA. Grenoble, juin 1986.

[144] PANDOLFI (Xavier) : *A Distributed Implementation of FP2's Communication's Mechanism.* Draft.

[145] PATERSON (M.S.), WEGMAN (Mark N.) : *Linear Unification.* Journal of Computer and System Sciences no. 16, pp. 158–167. Academic Press, 1978.

[146] PEREIRA-FERNANDEZ (Juan Manuel) : *Processus communicants : un langage formel et ses modèles. Problèmes d'analyse.* Thèse de 3ème cycle. Grenoble, juin 1984.

[147] PERRIN (Guy-René) : *Programmation parallèle : point de vue sur les langages et les méthodes.* TSI. Vol. 6, n° 2, pp. 103–113. Dunod, mars-avril 1987.

[148] PEYTON JONES (Simon L.) : *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[149] PLAICE (John Alexander) : *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone.* Thèse de l'INPG. Grenoble, mai 1988.

[150] RÉMY (Jean-Luc) : *Étude des Systèmes de Réécriture Conditionnels et Applications aux Types Abstraits Algébriques.* Thèse d'État. Nancy, juillet 1982.

[151] RÉMY (Jean-Luc), COMON (Hubert) : *How to Characterize the Language of Ground Normal Forms.* INRIA, RR no. 676. June 1987.

[152] REYNAUD (Jean-Claude) : *Sémantique de LPG.* RR 651–I–IMAG–56 LIFIA. Grenoble, mars 1987.

[153] ROBINSON (J.A.) : *A Machine-Oriented Logic Based on the Resolution Principle.* JACM. Vol. 12, no. 1, pp. 23–41. January 1965.

[154] ROGÉ (Sylvie) : *Comparaison des comportements des processus communicants. Application au langage FP2.* Thèse de l'INPG. Grenoble, novembre 1986.

[155] ROGÉ (Sylvie) : *An FP2 Parallel Interpreter.* In: "ESPRIT-Conference 88". North-Holland, 1988.

[156] RUSINOWITCH (Michaël) : *Démonstration automatique par des techniques de réécriture.* Thèse d'État. Nancy, novembre 1987.

[157] SCHAEFER (Peter), SCHNOEBELEN (Philippe) : *Specification of a Pipelined Event Driven Simulator Using FP2.* Proc. PARLE'87. Vol. I: Parallel Architectures. Eindhoven, June 1987. LNCS no. 258, pp. 311–328. Springer-Verlag.

[158] SCHMIDT-SCHAUß (Manfred) : *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Thesis. Kaiserslautern, April 1988.

[159] SCHNOEBELEN (Philippe) : *Sémantique du parallélisme en FP2.* Rapport de DEA. Grenoble, juin 1985.

[160] SCHNOEBELEN (Philippe) : *The Semantics of Concurrency in FP2.* R.R. IMAG 558 (LIFIA 30). Grenoble, October 1985.

[161] SCHNOEBELEN (Philippe) : *μFP2. A Prototype Interpreter for FP2.* R.R. IMAG 573 (LIFIA 41). Grenoble, January 1986.

[162] SCHNOEBELEN (Philippe) : *About the Implementation of FP2.* R.R. IMAG 574 (LIFIA 42). Grenoble, January 1986.

[163] SCHNOEBELEN (Philippe) : *Rewriting Techniques for the Temporal Analysis of Communicating Processes.* Proc. PARLE'87. Vol. II: Parallel Languages. Eindhoven, June 1987. LNCS no. 259, pp. 402–419. Springer-Verlag.

[164] SCHNOEBELEN (Philippe) : *Refined Compilation of Pattern-Matching for Functional Languages.* SCP 11 (1988), pp. 133–159. North-Holland. (Abridged version presented at an International Workshop. Gaussig (GDR), November 1988. In: "Algebraic and Logic Programming", pp. 233–243. Band 49. Akademie-Verlag, Berlin, GDR.)

[165] SCHNOEBELEN (Philippe), JORRAND (Philippe) : *Principles of FP2: Term Algebras for Specification of Parallel Machines.* In: "Languages for Parallel Architectures: Design, Semantics, Implementation Models". Wiley, to appear.

[166] SCHOENFIELD (J.R.) : *Mathematical Logic.* Addison-Wesley, 1967.

[167] SCOTT (Dana) : *Logic and Programming languages.* CACM. Vol. 20, no. 9, pp. 634–641. 1977.

[168] SMOLKA (Gert), NUTT (Werner), GOGUEN (Joseph A.), MESEGUER (José) : *Order-Sorted Equational Computation.* Proc. of the Colloquium on Resolution of Equations in Algebraic Structures. May 1987.

[169] SRIDHAR (S.) : *An Implementation of OBJ2: an Object-Oriented Language for Abstract Program Specification.* Proc. Foundations of Software Technology and Theoretical Computer Science. New Delhi, December 1986. LNCS no. 241, pp. 81–95.

[170] STEELE JR (Guy Lewis) : *COMMON LISP. The Language.* Digital Press, 1984.

[171] STRACHEY (Christopher) : *Fundamental Concepts in Programming Languages.* Lecture Notes from International Summer School in Computer Programming. Copenhagen, August 1967.

[172] THATCHER (James W.), WAGNER (Eric G.), WRIGHT (J.B.) : *Data Type Specification: Parameterization and the Power of Specification Techniques.* ACM TOPLAS. Vol. 4, no. 4, pp. 711–732. October 1982.

[173] TURNER (David A.) : *The Semantic Elegance of Applicative Languages.* Proc. of the 1981 Conference on Functional Programming Languages & Computer Architecture. Portsmouth (New Hampshire), October 1981. ACM, pp. 85–92.

[174] TURNER (David A.) : *Miranda: A Non Strict Functional Language with Polymorphic Types.* Proc. Functional and Programming Languages and Computer Architecture. Nancy, September 1985. LNCS no. 201, pp. 1–16. Springer-Verlag.

[175] VERJUS (Jean-Pierre) : *La recherche publique française sur le parallélisme et la répartition : le programme C^3.* TSI. Vol. 6, n° 2, pp. 201–208. Dunod, mars-avril 1987.

[176] VITTER (Jeffrey Scott), SIMONS (Roger A.) : *New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems for $\mathcal{P}$.* IEEE Transactions on Computers. Vol. C–35, no. 5, pp. 431–440. May 1986. (Extended abstract in: Proc. ACM'84 Annual Conference: The 5th Generation Challenge, pp. 75–84. October 1984.)

[177] WILLIAMS (John H.) : *On the Development of the Algebra of Functional Programs.* ACM TOPLAS. Vol. 4, no. 4, pp. 733–757. October 1982.

[178] YEMINI (Shaula), BERRY (Daniel M.) : *An Axiomatic Treatment of Exception Handling in an Expression-Oriented Language.* ACM TOPLAS. Vol. 9, no. 3, pp. 390–407. July 1987.

[179] YUASA (Taiichi), HAGIYA (Masami) : *Kyoto Common Lisp Report.* Teikoku Insastu Inc.

[180] ZHANG (Hantao), RÉMY (Jean-Luc) : *Contextual Rewriting.* Proc. RTA'85. Dijon, May 1985. LNCS no. 202, pp. 46–62. Springer-Verlag.

[181] — : *Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International organization for standardization. ISO/TC 97/SC 21 N. Project 97.21.20.2, July 1987.

[182] — : *The Programming Language ADA Reference Manual.* American National Standards Institute, Inc. ANSI/MLI–STD–1815A–1983. February 1983. LNCS no. 155. Springer-Verlag.

[183] — : *2.0 Sun Common Lisp Manual Set.* 825–1004–01. Sun Microsystems, Inc. Lucid, Inc. 1986.

Index

absorption 120
    règle d'— 120
adjoint
    — à droite 185
    — à gauche 185
adjonction 185
    unité de l'— 185
algèbre
    — d'erreur 128
    — des termes
        — satisfaisant un ensemble d'équations 15
        — sur une signature avec sortes ordonnées 22
        — sur une signature multi-sortes 11
    — hétérogène 10
    — libre 11
    — partielle 127
    — triviale 118
    support d'une — 10, 29
    Σ-— 10, 21, 29
arité
    — d'un opérateur 10
    — d'un prédicat 18
assignation 11, 18
atome 17

catégorie 177
    — de catégories 182
    — de foncteurs 183
    — des petits ensembles 178
    — terminale 55
    grande — 182
    petite — 182
classe 178
codomaine
    — d'un opérateur 10
composition
    — de flèches 177
    — de foncteurs 182
    — de présentations paramétrées 61
condition
    — d'une équation conditionnelle 18
    — en LPG 1.8 124
        — de déclenchement 124
        — de récupération 124
congruence 15, 18
constructeur
    — d'états 39
    opérateur — 32
        discriminant d'un — 84
        sélecteurs d'un — 84
construction libre 183
contexte
    — d'une équation conditionnelle 18
contrainte de sorte 28
    — valide 28
couple d'instanciation 70

"décompilation" 88
diagramme commutatif 178
domaine
    — d'un opérateur 10
    — d'un prédicat 18
    — d'une substitution 13
    — de définition 119

échappement (en Lisp) 143
endomorphisme 13
enrichissement 33
ensemble
    petit — 178
équation 14, 24
    — conditionnelle 18
        — valide 18
        prémisse d'une — 18
    — valide 14, 25
    Σ-— 14, 24

exception
déclenchement d'— 119
—-algèbre 130
—-morphisme 130
—-spécification 129
opérateur d'— 118
récupérateur d'— 121
récupération d'— 119
terme d'— 119
expression
— complète 103
— de modèle 65
— primitive d'une propriété 105
extension
— d'une présentation 32
— complète 33
— conservative 33
— consistante 33
— d'un foncteur libre 60

failures (en ML) 124
filtre 14
flèche 177
but d'une — 177
source d'une — 177
foncteur 182
— adjoint 185
— d'oubli 182, 186
— de synthèse 186
— libre 184

héritage 66
homomorphisme
— d'évaluation 186
Σ-— 10, 21, 30

implantation 82
— abstraite 82, 90
instance
— d'un terme 12
— d'une unité générique 66
instanciation 36, 52
couple d'— 70
— explicite 108
— simplifiée 100
— implicite 99, 108
interprétation 10, 18
isomorphisme 179
— de catégories 182
— naturel 183

lâche
sémantique — 56

modèle
déclaration de — 35
langage des expressions de —s 104
— d'un ensemble d'équations 14
— exigé 68
— générique 38
modules 33
morphisme 178
— de présentations 54
— de signatures 53

objet 177
— initial 179
—-Lisp 89
— terminal 179
opérateur 10
arité d'un — 10
codomaine d'un — 10
domaine d'un — 10
langage des expressions d'—s génériques 104
— complètement défini 31
— constant 10
— constructeur 32
— d'exception 118
— entre processus 41
— *safe* 129
— simulable par une présentation de processus 150
— *unsafe* 129
— variable 104
précondition d'un — 119
profil d'un — 10
ordre
— bien fondé 17
— de Kamin et Lévy 127
— lexicographique sur les chemins 127

paramètres formels 35
passage de paramètre
— générique 61

— standard 58
persistance 56
— forte 56
précondition
— d'un opérateur 119
— d'une règle de transition 39
règle de — 119
précongruence 15
prédicat 17
arité d'un — 18
domaine d'un — 18
prémisse
— d'une équation conditionnelle 18
préordre 178
présentation
— algébrique 14
morphisme de présentations 54
— complète par rapport aux booléens 19
— consistante par rapport aux booléens 19
— avec erreurs 128
— avec exceptions 131
— cible 54
— de processus 40
— paramètre 54
— paramétrée 54
correction d'une — 55
forte — 55
— protégée 32, 187
— structurée 31
postcondition
— d'une règle de transition 39
processus 39
connecteurs d'un — 39
histoire possible d'un — 40
message d'un — 39
présentation de — 40
opérateurs 41
abstraction 44
choix contrôlable 45
choix incontrôlable 45
composition asynchrone 45
composition parallèle 42
composition synchrone 45
connexion 43
contrôle par événement 46
copie indicée 45
déclenchement 46
time-out 46
— clavier (*KEYBOARD*) 217
— écran (*SCREEN*) 217
— terminé 40
signature de — 39
vecteur de — 141
produit fibré 181
profil d'un opérateur 10
propriété 35, 65
— exigée 36, 67
signature de — 35

réécriture
— équationnelle 24
règle de — 16
système de — 16
règle
— de précondition 119
— de réécriture 16, 24
— conditionnelle 19
— qui décroît les sortes 25
— de transition 39
postcondition d'une — 39
précondition d'une — 39
produit de deux règles de transition 42
— applicable 40
— immobile 42
— pré-applicable 40
— initiale 39
produit de deux règles initiales 42
renommage 13
rétraction 27, 179

satisfaction 66
algèbre des termes satisfaisant un ensemble d'équations 15
algèbre satisfaisant un atome 18
algèbre satisfaisant une présentation 14
signature
— algébrique
morphisme de signatures 53
— avec sortes ordonnées 21
composantes connexes d'une — 23
— cohérente 23

— régulière 22
— complètement habitée 11
— multi-sorte 10
— d'une propriété
— de processus 39
— de propriété 35
— ordonnée 65
somme amalgamée 59, 180
sorte 10
contrainte de — 28
langage des expressions de —s génériques 104
— avec égalité 207
— *Bool* 10
— habitée 11
— principale 113
— protégée 187
— variable 104
substitution 13, 105
domaine d'une — 13
stabilité par — 15
— fermée 15, 106
variables introduites par une — 13
surcharge 25, 37
système
— communicant 41
— de réécriture 16
équivalence par rapport aux constructeurs 208
propriété de Church-Rosser 17
sous-ensembles de factorisation 210
— confluent 16
— par rapport aux constructeurs 216
— convergent 17
— nœthérien 16
— qui décroît les sortes 25
— de transition 41

terme 11
instance d'un — 12
occurrences d'un — 12
plus fine levée d'ambiguïtés d'un — 26
positions d'un — 12
profondeur d'un — 12
racine d'un — 12
sous-—
— à une occurrence 12
— propre 32
— d'événement 39
— d'état 39
produit de deux termes d'état 42
— d'exception 119
— de communication 39
— en forme normale 16
— exceptionnel 120
— fermé 11
— irréductible 16
— linéaire 12
théorie 74
transformation 183
— naturelle 183
— universelle 184
type 15
module de — 33

unificateur
plus petit — 13
unification 13, 106
univers 178

validité
— d'une contrainte de sorte 28
— d'une équation 14, 25
— d'une équation conditionnelle — 18
variable 11
scission d'une — 208
— de travail 103
vue 77

$abs(\overline{\Omega})$ 132
$\mathbf{ALG}_{\mathcal{P}}$ 15
$\mathbf{ALG}_{\Sigma}$ 10
$\mathbf{ALG}_{\Sigma/\equiv_E}$ 15
Arr $\mathcal{C}$ 177
$At_{\Sigma}(V)$ 17
$Aux(t)$ 211
A^X 183
${\mathcal{C}_2}^{\mathcal{C}_1}$ 183
$\mathcal{C}(A, B)$ 177
Cat 182
Cat′ 182
Cls 178
couples 110
$cp(t)$ 85

cp_{RM} 92
CS 41
$D(\sigma)$ 13
decompile(x, s) 88
depth(t) 12
Ens$_{\mathcal{U}}$ 178
$ev => P$ 46
$ev \rightarrow P$ 46
$eval_{FP2}$ 72
$\mathrm{Ext}(F, h)$ 60
$f : A \cong B$ 179
`fm` 91
$\mathcal{G}^{=}$ 132
ge_{RM} 92
Grp 182
$h^{\sharp}$ 53
$I(\sigma)$ 13
id_A 177
$id_{\mathcal{C}}$ 182
implicit-instantiation 110
is-exceptional(t) 120
$LD(t)$ 26
$LS(t)$ 23
Mon 182
!*nothing-matches* 123
Ob $\mathcal{C}$ 177
occ(t) 12
OSA$_{\Sigma}$ 21
OSA$_{\Sigma/\equiv_E}$ 25
OSA$_{\Sigma/\equiv_E/C}$ 28
$\mathfrak{P}(u)$ 178
$P + A.B$ 43
$P - A$ 44
$P_1 \; ! \; P_2$ 45
$P_1 \; ? \; P_2$ 45
$P_1 \; \% \; n \; \% \; P_2$ 46
$P_1 \; || \; P_2$ 42
$P_1 \; ||| \; P_2$ 45
$P_1 \mid P_2$ 45
PP 40
Pre$_{(E,\preceq)}$ 178
Pres 54
rec(E) 132
root(t) 12
Set 178
Sig 53
Set$_S$ 184
solve-term 109
sort(t) 11
sort-constraint(c) 132
split 208
Subst$(T_{\Sigma}(V))$ 13
Subst$_{\mathbf{g}}(T_{\Sigma}(V))$ 15
$T_{\mathcal{P}}(V)$ 15
T_{Σ} 11
$T_{\Sigma}(V)$ 11
$\mathcal{T}_{\Sigma}(V)$ 22
$T_{\Sigma/\equiv_E}(V)$ 15
$\mathcal{T}_{\Sigma/\equiv_E}(V)$ 25
T_{Π} 39
$t[\tau \leftarrow t']$ 12
$t \xrightarrow{1}_{\mathcal{R}}$ 121
unique-term 109
var(t) 11
λ 12
$\tau : F \cong G$ 183
$\tau : F \dot{\rightarrow} G$ 183
$\underline{\omega}$ 11
$\overline{\omega(t_1, \ldots, t_n)}$ 11
1 55
$\models$ 14, 18, 28
$\diamond$ 62
$\uplus$ 186, 213
$\setminus$ 24
$\equiv$ 13
$\equiv_E$ 15
$==$ 14
$\|_c$ 42
$\longrightarrow_{\mathcal{R}}$ 16
$\langle F, G, \phi \rangle : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ 185
$\overset{\sigma}{\precsim}$ 14
$\approx$ 13
$\overset{*}{\longleftrightarrow}_{\mathcal{R}}$ 17
$\overset{*}{\longrightarrow}_{\mathcal{R}}$ 16
$\overset{\sim}{\longrightarrow}_{S}$ 211

Liste des exemples en FP2

0.80 Type *Bool* des booléens . . . 33
0.81 Type *Nat* des entiers naturels et égalité . . . 34
0.83 Propriété *Equality* (introduction informelle) . . . 35
0.85 Opérateur d'égalité générique à trois arguments . . . 36
0.86 Propriété *Ftype* . . . 36
0.87 Type des séquences linéaires . . . 36
0.88 Égalité des séquences génériques . . . 37
0.92 Processus d'addition arithmétique . . . 40
0.95 Multiplication par 2 (réseau de deux processus) . . . 44
0.96 Processus générique de duplication . . . 46
0.97 Processus réalisant une loi de composition interne . . . 46
1.23 Propriété d'égalité (rappel) . . . 65
1.24 Ordre partiel et ordre total . . . 66
1.25 Ordre lexicographique sur les séquences génériques . . . 68
3.2 Opérateur *alpha* de FP . . . 102
3.8 Composition d'un opérateur avec lui-même . . . 112
4.1 Type des rationnels positifs . . . 119
4.2 Division par 0 et récupération . . . 122
4.12 Exceptions et processus communicants . . . 142
B.1 Propriété *Ftype* (rappel) et opérateur d'identité . . . 189
B.2 Type *Bool* (rappel) . . . 190
B.3 Type *Nat* (version complète) . . . 191
B.4 Type *Seq* (version complète) . . . 192
B.5 Propriété d'égalité — Égalité des séquences . . . 192
B.6 Relations d'ordre — Tri rapide (*quicksort*) — Ordre lexicographique sur les séquences 193
B.7 Programmation "à la FP" . . . 194
B.8 Nombre d'apparitions d'un élément dans une séquence . . . 195
B.9 Somme disjointe de deux types . . . 196
B.10 Produit cartésien de deux types . . . 196
B.11 Exemple d'instanciation partielle : "carré" d'un type. . . . 197
B.12 Propriétés d'une loi de composition interne . . . 197
B.13 Arbres à nombre quelconque de fils . . . 197
B.14 Arbres quelconques d'entiers . . . 199
B.15 Définition d'utilitaires divers pour un type ordonné . . . 200
B.16 Arbres binaires ordonnés . . . 201
B.17 Type *Bounded-Stack* des piles bornées génériques . . . 201

B.18 Allocation de mémoire, après récupération d'une exception 202
B.19 Définition d'un isomorphisme à partir d'un opérateur de conversion 202
B.20 Type *Char* des caractères 203
B.21 Type *String* des chaînes de caractères 203
D.1 Session avec entrées-sorties 219
D.2 Remplacement d'entrées entières erronées pour cause d'échec des analyses syntaxique ou sémantique 220
D.3 Remplacement de toutes les entrées entières qui déclenchent une exception 220

Table des matières

Introduction **3**

0 Notions de base — Présentation succinte de FP2 **9**

0.1 Programmation fonctionnelle 9
0.1.1 Les bases théoriques de la spécification algébrique 9
0.1.1.1 Signatures — Algèbres hétérogènes 10
0.1.1.2 Termes 11
0.1.1.3 Équations 14
0.1.1.4 Réécriture : les bases 15
0.1.1.5 Réécriture conditionnelle 17
0.1.2 Une généralisation : les algèbres avec sortes ordonnées 20
0.1.2.1 Signatures avec sortes ordonnées — Modèles sémantiques de Joseph Goguen et José Meseguer 21
0.1.2.2 Algèbre des termes 22
0.1.2.3 Équations et règles de réécriture 23
0.1.2.4 Rétractions 27
0.1.2.5 Contraintes de sortes 28
0.1.2.6 Une autre approche : sémantique de Gert Smolka 29
0.1.2.7 Brève comparaison de ces deux approches 30
0.1.3 Présentations structurées 31
0.1.4 Importation de définitions 32
0.1.4.1 Notions générales 32
0.1.4.2 Conventions de FP2 33
0.1.4.3 Directives d'OBJ3 — Comparaison avec FP2 34
0.1.5 Introduction informelle à la généricité en FP2 35
0.1.6 Portée des objets en FP2 : tableau récapitulatif 38
0.2 Programmation parallèle 38
0.2.1 Processus 39
0.2.1.1 Le niveau syntaxique 39
0.2.1.2 Le niveau sémantique 41
0.2.2 Opérateurs entre processus 41
0.2.2.1 Composition parallèle 42
0.2.2.2 Connexion 43
0.2.2.3 Abstraction 44
0.2.2.4 Copie indicée 45
0.2.2.5 Autres opérateurs 45

0.2.2.5.1 Composition asynchrone 45
0.2.2.5.2 Composition synchrone 45
0.2.2.5.3 Choix incontrôlable 45
0.2.2.5.4 Choix contrôlable 45
0.2.2.5.5 Déclenchement 46
0.2.2.5.6 Contrôle par événement 46
0.2.2.5.7 *Time-out* 46
0.2.3 Processus et généricité 46

I Généricité **49**

1 Les bases de la généricité **51**
1.1 Théorie de la généricité 52
1.1.1 Morphismes de signatures — morphismes de présentations 53
1.1.2 Présentations paramétrées 54
1.1.3 Passage de paramètre standard 57
1.1.4 Passage de paramètre générique 61
1.1.5 Composition de paramétrisations 62
1.2 Généricité en FP2 65
1.2.1 Présentations en paramètre 65
1.2.2 Relations entre propriétés 66
1.2.3 Présentations paramétrées 67
1.2.4 Morphismes de présentations 69
1.2.5 Modèles génériques 69
1.2.6 Passage de paramètres 70
1.2.7 Sémantique opérationnelle 72
1.3 Généricité en OBJ3 74
1.3.1 Théories 74
1.3.2 Objets paramétrés 75
1.3.3 Morphismes de présentation 77
1.3.4 Comparaison 78

2 Réalisation **79**
2.1 Principes généraux 80
2.1.1 Boucle d'évaluation de FP2 80
2.1.2 Compilation des sortes 82
2.1.2.1 Approche générale 82
2.1.2.2 Approche de FP2 83
2.1.2.3 Propriétés de correction 85
2.1.3 Compilation des opérateurs 87
2.1.4 "Décompilation" — Traitement des formats externes spéciaux 88
2.2 Compilation des termes génériques 90
2.2.1 Un résultat théorique 90
2.2.2 Règles 91
2.2.3 Simulation des opérateurs génériques par des fonctions du second ordre 93

3 Raccourcis de notation **99**
3.1 Approche intuitive 99
3.1.1 Instanciation implicite 99
3.1.2 Instanciation explicite simplifiée 100
3.1.3 Analyse de la sorte d'une expression — Ambiguïtés 101
3.1.4 Conclusion de l'approche intuitive 103
3.2 Conditions d'utilisation de la surcharge 103
3.3 Définition du formalisme 104
3.3.1 Syntaxe 104
3.3.2 Opérations 106
3.3.3 Sémantique 107
3.3.4 Implantation 107
3.4 Le problème posé 107
3.5 Règles d'analyse 108
3.5.1 Prise en compte de la surcharge 109
3.5.2 Calcul d'instanciation implicite 110
3.5.3 Critères et preuves 111
3.5.3.1 Correction 111
3.5.3.2 Complétude 112
3.5.4 Terminaison 112
3.6 Raccourcis de notation en OBJ3 — Comparaison 113

II **115**

4 Exceptions **117**
4.1 Première approche des cas d'erreur 117
4.2 Description opérationnelle des exceptions 118
4.2.1 Règles de précondition 119
4.2.2 Déclenchement d'exception dans une évaluation 120
4.2.3 Récupération d'exception 121
4.2.4 Exceptions avec paramètres formels 122
4.2.5 Homogénéisation 123
4.2.6 Remarque importante 123
4.2.7 Réalisations voisines 124
4.2.7.1 Conditions de LPG 1.8 124
4.2.7.2 *Failures* en ML — Utilisation en LCF 124
4.3 Exceptions et propriété de terminaison 125
4.4 Comparaison avec d'autres concepts 127
4.4.1 Algèbres partielles 127
4.4.2 Sous-sortes 127
4.4.3 Opérateurs "OK" et opérateurs "erreur" 128
4.4.4 Opérateurs *safe* et opérateurs *unsafe* 129
4.4.5 Exception-spécifications et exception-algèbres 129
4.4.6 Algèbres avec sortes ordonnées 130
4.5 Sémantique algébrique 131
4.5.1 Présentation avec exceptions : cas particulier de présentation avec sortes ordonnées 131

4.5.2 Propriétés de la présentation obtenue — Niveau sémantique 135
4.5.3 Niveau opérationnel 137
4.5.4 Remarques et compléments 137
4.5.4.1 Introduction de nouvelles définitions 137
4.5.4.2 Exceptions et généricité 137
4.5.4.3 Conditions et termes booléens 139
4.5.4.4 Exceptions et sémantique de Gert Smolka 140
4.5.5 Conclusion de l'étude sémantique 140
4.6 Traitement local des exceptions dans les processus FP2 140
4.6.1 Nécessité d'intégrer un mécanisme d'exceptions 140
4.6.2 Introduction des exceptions — Conséquences 141
4.7 Implantation des exceptions 143
4.7.1 Échappements 143
4.7.2 Implantation des exceptions de FP2 en Common Lisp 144

III **147**

5 Transformation de programmes **149**
5.1 Les idées de base 149
5.1.1 Introduction informelle 149
5.1.2 Cadre de l'étude 150
5.2 Hypothèses — Définitions fonctionnelles nécessaires 151
5.2.1 Les hypothèses 151
5.2.2 Type "zéro ou un" 153
5.3 Transformation : la démarche 154
5.3.1 Parallélisme en largeur : le réseau obtenu 154
5.3.2 Les termes trop profonds 156
5.3.3 Pipe-line 157
5.4 Preuve de correction 158
5.5 Discussion — Améliorations — Variantes 162
5.5.1 Discussion 162
5.5.2 Parallélisation à l'intérieur de H 163
5.5.3 Abaissement du nombre des processus 164
5.6 Élimination des évaluations redondantes 164
5.6.1 L'idée de base 164
5.6.2 Réseau en couronne 167
5.6.3 Optimisations possibles 168
5.6.3.1 Minimisation des échanges de messages 168
5.6.3.2 Utilisation de la symétrie de la connexion 168

Conclusion **171**

Annexes **175**

A Rappels de théorie des catégories **177**
A.1 Catégories 177
A.1.1 Définitions de base 177
A.1.2 Définitions diverses 179
A.1.3 Constructions catégoriques 179
A.1.3.1 Somme amalgamée 179
A.1.3.2 Produit fibré 181
A.2 Foncteurs — Transformations — Adjonction 181
A.2.1 Foncteurs 181
A.2.2 Transformations 183
A.2.3 Construction libre 183
A.2.4 Adjonction 184
A.2.5 Interprétation catégorique des importations de définitions 186

B Exemples **189**
B.1 Propriétés et types "de base" 189
B.2 Exemples divers 194
B.2.1 Programmation "à la FP" 194
B.2.2 Structures algébriques 195
B.2.3 Arbres 197
B.2.4 Utilisation d'exceptions 201
B.2.5 Définition des caractères et des chaînes de caractères par transfert de structure 202
B.3 Exemples de codes générés par le compilateur FP2 204

C Linéarisation d'un système de réécriture **207**
C.1 Quelques définitions techniques 207
C.2 Linéarisation d'un système de réécriture 209
C.3 Résultats de confluence 216

D Entrées-sorties en FP2 **217**
D.1 Processus "clavier" et "écran" 217
D.2 Réalisation — Exemples 218
D.3 Récupération d'entrées erronées 219

Bibliographie **223**

Index **235**

Liste des exemples en FP2 **241**

département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

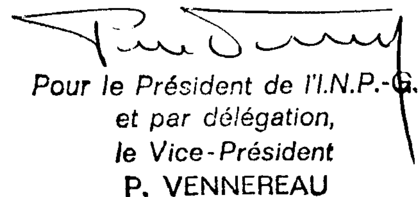
VU les rapports de présentation de Messieurs

- J.P. BANATRE , Professeur
- Cl. KIRCHNER , Directeur de Recherche

Monsieur HUFFLEN Jean-Michel

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " **Informatique** "

Fait à Grenoble, le 27 juin 1989



Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU

Résumé

FP2 (**F**unctional **P**arallel **P**rogramming) est un langage qui concilie programmation fonctionnelle et programmation parallèle à travers le formalisme des spécifications algébriques et des systèmes de réécriture. Dans le cadre du projet FP2, cette thèse a pour principal objectif de présenter la partie fonctionnelle, incluant la généricité et le traitement des exceptions.

La généricité (paramétrisation d'une spécification) est traitée dans la première partie : nous rappelons les principes, étudions la sémantique, formalisons la compilation des opérateurs génériques en restant dans un cadre fonctionnel, et analysons les raccourcis de notation offerts aux utilisateurs.

La deuxième partie est consacrée aux exceptions. Elles sont d'abord étudiées d'un point de vue opérationnel, puis nous en donnons une définition précise qui permet de ramener une présentation avec exceptions à une présentation avec sortes ordonnées. Cette définition assure l'existence d'une algèbre initiale et permet en outre de traiter les exceptions avec paramètres génériques.

En troisième partie, sont présentées des méthodes de transformation de définitions fonctionnelles récursives en processus parallèles communicants. La généricité est utilisée pour formuler les hypothèses sur les définitions fonctionnelles, et nous montrons de plus comment simuler une pile de récursivité de profondeur arbitraire par des réseaux de processus dont la topologie est fixée statiquement.

Mots-clés : spécification algébrique, langage fonctionnel et parallèle, généricité, sémantique, compilation, exceptions, transformation de programmes.

Abstract

FP2 (**F**unctional **P**arallel **P**rogramming) is a language combining both functional and parallel programming through the algebraic specification approach and term rewriting systems. As part of the project FP2, the principal object of this thesis is the presentation of the functional part, including genericity and the treatment of exceptions.

Part I presents genericity (i.e. parameterization of a specification). We recall the principles, study the semantics, formalize the compilation of generic operators in a functional framework, and analyze the notational facilities provided to the users.

In part II, we investigate the exception mechanism. At first, we present them operationally. Then, we show that it is possible to define a presentation with exceptions as an order-sorted presentation. Thus, the existence of an initial algebra is ensured. Moreover, this method permits to treat exceptions with generic parameters.

In part III, methods for transforming recursive functional definitions into parallel programs are presented. The hypotheses on the functional definitions are specified by means of generic operators. Besides, we show how to simulate an arbitrarily deep recursive stack by means of networks with a statically fixed topology.

Keywords: algebraic specification, functional and parallel language, genericity, semantics, compilation, exceptions, program transformation.