Conception et implantation d'un modèle de raisonnement sur les contextes basée sur une théorie des types et utilisant une ontologie de domaine

Patrick Martial, Joseph Barlatier

Département de formation doctorale en informatique **École doctorale SISEO**
Polytech'Savoie

Conception et implantation d'un modèle de raisonnement sur les contextes basé sur une théorie des types et utilisant une ontologie de domaine

THÈSE

présentée et soutenue publiquement le 16 juillet 2009

pour l'obtention du

Doctorat de l'Université de Savoie

(spécialité informatique)

par

Patrick Barlatier

Composition du jury

Président :	Amédéo Napoli
Rapporteurs :	Jean-Pierre Desclés Jérôme Euzenat
Examinateurs :	Christophe Roche (Directeur de thèse) Richard Dapoigny (Co-directeur de thèse) Patrick Brézillon Christophe Raffalli
Invité :	David Menga

LISTIC

Mis en page avec la classe thloria.

Remerciements

Quelle étincelle allume le feu de l'imagination ? La naissance d'une idée est un processus dont nul ne connaît l'exact cheminement mais qui prend source dans l'échange avec les autres et la conversation parfois contradictoire.

Nombreuses seront donc les personnes qui mériteraient de figurer dans ce présent hommage et que je ne citerai pas puisque c'est au cours d'une multitude de conversations, de la plus simple à la plus sérieuse, que, durant les trois années de gestation de ma thèse, les idées ont germé pour aboutir à ce travail.

Cependant, sans un environnement scientifique stimulant, je n'aurais même pas eu le commencement d'une idée. Je tiens donc à témoigner ma reconnaissance tout d'abord à mes directeurs de thèse : Christophe Roche et Richard Dapoigny. L'incroyable disponibilité de Richard et sa patience dans les moments difficiles m'ont permis de finaliser ce travail. J'aurai également une pensée pour Eric Benoit qui m'a toujours encouragé et pour Claude Genier du Conservatoire National des Arts et Métiers qui m'a suivi et aidé (ainsi que beaucoup d'autres) tout au long de mes longues années d'études dans son établissement à Saint-Genis.

Je témoigne également ma reconnaissance aux rapporteurs : Jérôme Euzenat et Jean-Pierre Desclés.

Je dédie cette thèse à Chantal Boinnard, la femme qui partage ma vie et à notre fils Johan, à ma famille et à mon ami Jacques Laty.

Table des matières

Introduction générale

" Nous ne jugeons des situations que parce que nous voyons les objets dans un lieu où ils occupent chacun un espace déterminé ; et nous ne jugeons du mouvement que parce que nous les voyons changer de situation, [Condillac. Traité des sens. I, II, 9.]. "

Depuis quelques années déjà, les informations se référant à un environnement donné (aussi bien pour décrire une réalité abstraite que concrète) sont de plus en plus exploitées par les systèmes informatiques, que ce soit dans le domaine de la linguistique où ces informations servent à déterminer le sens d'un nom (p. ex., les méta-informations sur un livre), ou que ce soit dans les applications traitant des environnements changeants, comme les téléphones portables, les GPS, etc., domaines dans lesquels ces données permettent entre autres choses [1] de choisir une action à entreprendre, de déterminer la nature des entités impliquées ou de s'adapter à un environnement.

Dans ce mémoire nous proposerons une solution possible à la question suivante : comment formaliser ces environnements et comment utiliser les informations qu'ils contiennent pour produire des actions pertinentes ?

La réalité s'exprime par la description d'un ensemble d'objets (choses, individus) généralement reliés entre eux et de leurs propriétés (une application industrielle, une habitation, un programme informatique, etc.). Cet ensemble d'objets qui témoigne d'une certaine réalité sera appelé par la suite " situation ". La description d'une situation peut être envisagée de deux manières : l'une en considérant des objets concrets, l'autre en décrivant ces objets de manière abstraite, ce qui forme un concept, qui par la suite sera appelé " type ". Dans ce dernier cas, les concepts et leurs relations peuvent être réutilisés dans toutes sortes de descriptions. A titre d'exemple, le type " personne " est une abstraction de Jean qui a comme numéro de sécurité sociale : 12-563-456... . La définition d'une situation abstraite devient alors plus précise et s'énonce maintenant comme étant une collection des concepts qui décrivent les objets du domaine, pour une communauté donnée, ce qui est une " description du réel ".

Pour des besoins de réutilisabilité, un système informatique de spécification des concepts et de leurs relations devient nécessaire et les modèles développés pour la représentation des connaissances, tels que les ontologies, répondent précisément à ce critère. Une ontologie peut être perçue comme un ensemble de concepts et de relations (entre ces concepts) définis à l'aide d'un langage formel compréhensible par un ordinateur. Cette définition permet donc de mettre en correspondance situation et ontologie, pour lier les éléments d'une situation et leurs relations avec leurs abstractions (types).

Les situations sont la représentation d'un état ponctuel du monde réel qui peut être modifié par toutes sortes d'événements. Parmi ces événements, l'action tient une place de choix pour la raison suivante : le concept d'action doit être interprété comme étant le moyen d'expression d'une intention. L'intention motive l'action et cette dernière transforme une situation en une autre situation selon un modèle classique en IA (le calcul des situations). La relation entre situation et action repose sur les observations suivantes : la situation n'implique pas l'action (relation non causale) mais par contre une action pourra avoir lieu dans une situation donnée. La relation doit donc plutôt être vue comme une association décrite par la paire : $<$ partie d'une situation, action $>$. Par ailleurs, le concept d'action dépend de l'état des objets contenus dans une situation pour des raisons diverses, par exemple : parce que les objets possèdent des propriétés utilisées dans des

1. Le typage permet de catégoriser les informations

calculs ou parce que les objets possèdent des propriétés qui seront modifiées ou encore parce que les relations entre les objets seront changées. En prenant l'exemple de l'action qui consisterait à rembourser une personne de ses frais médicaux, une donnée associée à et dépendante de cette action serait le numéro de sécurité sociale de cette personne.

Les objets, relations et valeurs dont une action dépend forment une collection d'informations qui sera appelée contexte de l'action. La dépendance entre les actions et leurs contextes peut être formalisée par la notion de fonction. Pour traduire simultanément les notions d'association et de dépendance fonctionnelle entre un contexte et une action, la paire (au sens mathématique du terme) sera utilisée dans son sens le plus strict (ordonnancement). Un contexte est associé à une action pour préserver la nature fonctionnelle de cette relation, alors qu'une action peut être associée à un ou plusieurs contextes.

Situation

Alfa-Roméo Bertone &
MI- 2051 &
95 km/h &
conducteur(patrick) &
passager(nathalie)

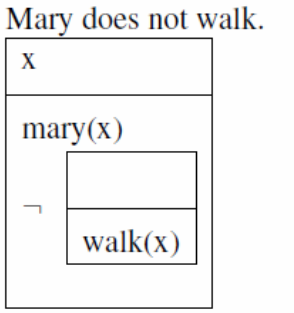
Contexte	Action
ivre(patrick)	intercepter(patrick) & verbaliser(patrick)
enceinte(nathalie)	escorter(patrick,hôpital)
retard(patrick,cours1)	intercepter(patrick)
ivre(patrick) & enceinte(nathalie)	intercepter(patrick) & conduire(nathalie,hôpital)

Dans l'exemple élémentaire ci-dessus, un agent de circulation reçoit des informations contextuelles pour prendre une décision correspondant à une action dépendant du contexte. En ajoutant aux éléments d'information peuplant la situation le fait que Patrick est ivre, alors l'action qui lui est associée sera d'intercepter le conducteur. En revanche, si l'information incorporée à la situation nous informe que la passagère est enceinte, alors l'agent devra les escorter jusqu'à l'hôpital le plus proche. Remarquons que tous les éléments de la situation ne sont pas forcément pris en compte dans le contexte et qu'une seule action peut être associée à plusieurs contextes (Patrick peut être soit ivre, soit en retard, voire les deux, et dans ce cas nous parlerons de composition de contexte). Par ailleurs, supposons maintenant que le contexte ivre(patrick) soit prouvé (et ajouté à la situation) et que le contexte enceinte(nathalie) soit également prouvé. Ce cas doit être envisagé en lui associant l'action « intercepter(patrick) & conduire(nathalie, hôpital) ». Il est important de noter que l'ajout d'information à un contexte peut déboucher sur un nouveau contexte valide qui est un sous-type du précédent et que, dans ce cas, le contexte correspondant à l'information la plus précise est sélectionné avec son action associée.

Les contextes constituent la pierre angulaire de cette recherche. En effet, en partant de l'hypothèse que le résultat d'une intention pourrait être le but à atteindre d'un agent logiciel, il apparaît nécessaire d'identifier des contextes vérifiés dans une situation donnée et de produire

les actions qui en dépendent. L'agent logiciel qui agirait de la sorte pourrait ainsi répéter cette tâche d'identification des contextes jusqu'à l'obtention du but recherché. Dans le domaine de la linguistique, le sens d'un mot ou d'une phrase tirée d'un livre par exemple est identifié par une structure de données. Le contexte est une partie d'une situation abstraite et sa vérification reprend les données abstraites des éléments d'une situation concrète (par conséquent des concepts de l'ontologie).

Ce travail s'inspire en partie des théories de l'information lexicale, où l'interprétation du sens d'un mot dans une phrase s'appuie sur un ensemble d'informations décrivant le contexte du mot qui est rassemblé dans des structures syntaxiques (e.g., les *Discourse Representation Structures* ou DRS d'Asher [4], les *Dependent Record Types* ou DRT de Cooper [35], ...). Considérons par exemple la structure suivante (DRS) qui représente la phrase Mary does not walk.



Les DRS sont représentées par des boites divisées en deux parties, une partie supérieure représentant l'univers du DRS et une partie inférieure représentant des conditions. L'aspect intéressant tient dans la récursivité possible de cette structure (les DRS peuvent contenir d'autres DRS subordonnées dans leurs conditions). Les Dependent Record Types (DRT) sont les éléments primitifs d'une théorie, dite théorie constructive des types. Ils sont capables de circonscrire un ensemble de connaissances et de contraintes d'une manière dynamique (extension possible des DRT) et de raisonner sur cette structure. Nous nous sommes inspirés simultanément de l'aspect récursif des DRS et de l'aspect extensionnel des DRT en utilisant une théorie constructive des types pour représenter et raisonner sur les contextes.

La théorie constructive des types possède un certain nombre d'avantages tout à fait adaptés à la modélisation du contexte. Tout d'abord, elle exploite une correspondance entre la logique constructive et des programmes informatiques dont l'avantage principal tient dans la calculabilité qui en résulte. Ensuite, à la différence de la logique du premier ordre, elle permet de quantifier correctement sur des sous-ensembles (des sous-types en théorie des types). Par exemple, en logique du premier ordre, la phrase *Every man who owns a donkey beats it* (« Tout homme qui possèdent un âne le bat ») se représente par la formule :

$$\forall d \in Donkey, \forall m \in Man(Owns(m,d) \supset Beats(m,d))$$

Le problème vient de ce que la quantification porte sur tous les ânes, au lieu de porter uniquement sur les hommes possédant un âne. Avec la théorie constructive des types, il est possible de quantifier uniquement sur ce « sous-ensemble » et l'exemple ci-dessus devient :

$$\Pi d : (\Sigma x : Man.(\Sigma y : Donkey.Owns(x,y)).(Beats(\pi_1 d, \pi_1(\pi_2 d))$$

Les opérateurs Σ produisent un ensemble d'objets caractérisant l'ensemble des hommes qui possèdent un âne. La quantification universelle (Π) porte alors uniquement sur cet ensemble (qui est bien un sous-ensemble de l'ensemble des hommes). Cette quantification universelle est dépendante au sens où la proposition résultante (Beats) dépend des objets de l'ensemble des hommes qui battent leur âne (objets référés ici par $\pi_1 d$ et $\pi_1(\pi_2 d)$). Ceci améliore considérablement la précision et la concision des expressions qu'il est envisageable de représenter. Contrairement à la théorie des ensembles, il est possible de faire la distinction entre les objets de l'ensemble et donc d'apporter d'avantage d'informations. D'autre part, la notion de type dépendant (i.e., des types qui dépendent de termes du langage [termes qui peuvent être des valeurs]) constitue un élément fondamental de l'accroissement d'expressivité nécessaire pour exprimer de manière simple mais précise des descriptions complexes de parties d'une situation. Enfin, un autre argument en faveur du choix d'une théorie constructive des types est qu'elle permet de réduire d'une manière significative la taille de l'espace des solutions d'un problème puisqu'en restreignant les ensembles de quantification on réduit le volume des informations à traiter ce qui diminue la complexité.

À partir d'une théorie constructive des types, notre approche en propose une extension afin de répondre aux problèmes du raisonnement sur les contextes et les actions. Cette extension, appelée Dependent Type Framework (DTF), étend la théorie de base en définissant la sémantique de deux nouveaux types : le contexte et l'action et en exploitant le sous-typage des contextes tout en conservant la décidabilité de la théorie de base.

Nous montrerons comment modéliser les contextes sous la forme de types DTF à partir des données fournies sur un problème et des actions à entreprendre pour le résoudre. A cette phase de conception, vient s'ajouter une phase d'exécution dans laquelle un programme informatique est capable, à partir d'une situation courante, d'identifier le ou les contextes vérifiés pour en déduire la ou les actions à envisager.

Enfin, pour tester la faisabilité et pouvoir juger de la complexité d'une telle solution, un " démonstrateur de contexte " est réalisé avec un langage fonctionnel. Puis, une application test appelée " le monde du Wumpus " dans lequel un agent logiciel se déplace dans un environnement inconnu, est alors modélisée.

Dans [116], les auteurs identifient quatre niveaux de représentation des connaissances, et ils en donnent une classification selon l'ordre de leur étude. Ce sont :

- Le niveau conceptuel qui est appelé sémantique, i.e., le sens que l'on donne aux connaissances représentées.
- Le niveau épistémologique qui est appelé syntaxique (primitives du langage).
- Le niveau logique.
- Le niveau implantation informatique, i.e., le codage et à la résolution algorithmique du problème et à sa mise en oeuvre par la programmation.

Nous proposerons donc dans ce mémoire l'étude d'une solution à la conception et l'implantation d'un modèle de représentation et de raisonnement sur les contextes en suivant cette classification.

Le niveau conceptuel sera abordé par la présentation des notions d'ontologies, de contextes et de types qui fournissent le niveau sémantique. Le niveau épistémologique et le niveau logique correspondent à la spécification d'un langage que nous avons appelé Dependant Type Frame-

work qui propose à la fois une syntaxe pour la représentation des concepts et des relations entre ces concepts et une logique d'ordre supérieur interne (intuitionniste). Enfin, nous donnerons un exemple d'implantation de ce langage, qui repose sur le λ-calcul.

Le mémoire est structuré de la manière suivante :

- La première partie introduit les principes préliminaires (i.e. à savoir les ontologies, la notion de contexte, la logique intuitionniste et la théorie des types) nécessaires à la compréhension de la solution proposée.
- La seconde partie décrit le cadre de travail proposé.
- Enfin, la troisième partie montre à travers un exemple une implantation possible.

Première partie

Ontologies, Contextes, Logiques et Théories des Types

Au fur et à mesure qu'une notion devient plus familière, il arrive que l'on néglige de plus en plus de remonter aux sources imagées par où sa signification s'est établie en l'esprit. Les nouvelles des espèces superposées à la précédente tendent à s'affranchir davantage du contrôle exercé par la pensée substantielle et à se confondre avec leur expression formelle. Cette erreur, tantôt prise en habitude ou tantôt évitée, marque les époques alternantes de stagnation ou de progrès de l'invention mathématique.

[Arnaud Denjoy]

1

De la notion d'ontologies

La gestion des connaissances est un besoin éprouvé de nos jours par de nombreux acteurs de domaines d'applications différents. Elle nécessite donc de plus en plus la définition d'un format d'échange commun à ces activités [2] : gestion de contenus et de documents sur le Web, coopération, communautés, construction d'outils sémantiques, tels sont quelques mots-clés d'une tendance à la mise en commun de "sens partagés" dans l'activité. Elle s'intéresse aux principes, aux modèles, aux théories, aux processus ou aux méthodes, notamment aux connaissances tacites des experts, ainsi qu'aux opérations et aux principes de décision qui en découlent.

Ces dernières années sont apparues de nombreuses techniques de découverte de connaissances par l'intermédiaire de la création, automatique ou semi-automatique, de structures relationnelles complexes de connaissances. Par exemple, un courant de recherche actuel très actif vise à construire un "Web sémantique" à partir d'un langage pour exprimer la connaissance reposant sur une sémantique décrite avec la syntaxe OWL et une syntaxe d'échange normative de OWL : RDF/XML. Il est essentiel pour capturer la sémantique associée aux mots à partir de textes ou encore à développer des mécanismes de recherche d'informations, à l'aide d'agents intelligents explorant le Web (Systèmes Multi-Agents).

Les bases de connaissances offrent une aide à l'identification pour ceux qui ne connaissent pas le nom d'un concept, mais également une aide à la classification pour les spécialistes dont le domaine n'est que partiellement étudié. Elles offrent la possibilité d'élaborer une véritable ontologie d'un domaine, à travers une modélisation fine des concepts, des descriptions, de la terminologie utilisée et du sens associé aux mots. Ces ontologies d'un domaine doivent être vues comme le moyen d'établir une base conceptuelle commune, dans le but de communiquer des connaissances relatives au domaine, dans un but précis.

1.1 Introduction informelle aux notions de base

Nous exposerons ici les principales notions qui seront formalisées par la suite. La notion centrale est celle de concept. Il est à remarquer que la plupart des schémas de représentations des connaissances, qu'ils soient formels ou informels, correspondent au triangle sémiotique (Peirce,

2. Le modèle relationnel souffre de limitations dues principalement au fait que par exemple il n'existe :

1. Aucune sémantique n'est associée aux données internes.

2. Aucune sémantique n'est associée aux relations internes et externes à la base de données.

Ajouter à cela qu'il est difficile de faire évoluer le modèle de données, décrit a priori, avant l'insertion des données.

1958)(voir 1.1). L'interprétation du triangle sémiotique est hors de portée de cette thèse et nous ne fournirons simplement qu'une explication intuitive expliquant nos choix dans la conception de notre formalisme.
Le triangle sémiotique est une manière de montrer la relation qui existe entre une abstraction (la référence ou une pensée) et son expression par des objets concrets (des référents) du monde réel et le symbole qui est utilisé pour le dénoter.

- **Objet** C'est à Frege que l'on doit la distinction entre objet (Gegenstand) et concept (Begriff). Ceci traduit deux points de vue :
  - Une opposition entre sujet et prédicat d'une proposition ; le Gegenstand fonctionnant nécessairement comme sujet (ce dont on dit quelque chose), et le concept étant le prédicat.
  - Une opposition entre symbole « saturé » et « non saturé », le symbole d'objet est saturé en ce qu'il est complet en lui-même, tandis que le symbole de concept (ou plus généralement de « fonction ») présente une ou plusieurs places encore vides et devant être occupées par des noms.

  Pour nous un objet doit être compris comme étant la représentation d'un objet du monde réel. Du point de vue des ontologies, il correspond à la notion d'« individu » et du point de vue logique il correspond aux preuves d'un terme.
- **Propriété** La propriété d'un objet (ou d'une collection d'objets) est une qualité ou une caractéristique d'un objet. Une propriété peut aussi être interprétée comme une entité abstraite. La propriété n'existe que dans les objets, non par elle-même ; elle n'est réelle que de la réalité même de la chose de laquelle elle est inhèrante. Du point de vue des ontologies, elle correspond à la notion d' « attribut ».
- **Fait** Un fait dénote des objets en relation.
- **Predicat** Un prédicat est une fonction qui retourne une valeur de vérité. Les prédicats sont des objets vrais qui expriment des propriétés ou des contraintes.
- **Classe** Une classe est une abstraction d'une collection d'objets qui possède des propriétés en commun. Du point de vue des ontologie, une classe est plus spécifiquement une définition formelle d'un ensemble de particuliers qui possède un ensemble donné d'attributs.
- **Terme** Un terme est un nom (terme simple), une expression (terme complexe), un symbole ou une formule qui désigne un objet pour un sujet donné.
- **Concept** Un concept est une unité de connaissances créée par une combinaison unique de propriétés. Suivant Carnap les concepts se caractérisent par un principe d'intensionnalité : deux concepts sont identiques si et seulement si tout individu qui exemplifie l'un, exemplifie aussi nécessairement l'autre. D'un point de vue ontologique, on ce réfère au triangle sémiotique (1.1) : le concept représente un objet qui est exprimé par un terme. Il s'ensuit que le sens d'un terme est dénoté par un concept.
- **Intension** L'intension est un ensemble de propriétés qui caractérise un concept.
- **Extension** L'extension décrit un ensemble d'objets qui tombent sous un concept. Un concept peut être caractérisé par son intension ou son extension.
- **Subsomption**
  En reprenant l'idée proposée par [104], il existe trois conceptions possibles de la subsomption :
  Définition 1. (Approche extensionnelle) Un concept X subsume un concept Y si et seulement si l'ensemble des instances de X est inclus dans l'ensemble des instances de Y.
  Définition 2. (Approche intensionnelle) Un concept X subsume un concept Y si tout ce qui est décrit par Y est aussi représenté par la description plus générale de X.

Définition 3. (Approche hybride) Un concept X subsume un concept Y si les définitions de X et de Y impliquent logiquement que toute instance de Y est également une instance de X.
Ces trois conceptions se distinguent par la manière dont les individus du domaine sont considérés. Avec l'approche hybride, la définition est plus proche de la démarche ontologique, alors que l'approche intensionnelle correspond au modèle orienté objet et l'approche extensionnelle est celle adoptée par les logiques de description [c.f. 3.2].

La subsomption est un ordre partiel : elle est réflexive (X subsume X), antisymétrique (si X subsume Y et Y subsume X, alors X et Y sont identiques) et transitive (si X subsume Y et Y subsume Z, alors X subsume Z). Les éléments reliés par la subsomption sont organisés par une taxonomie.

Dans ce mémoire nous utiliserons une approche hybride.
- **Proposition** Les propositions sont des formules décrivant des propriétés et des faits.
- **Jugement** Les jugements sont la possibilité de pouvoir prouver les formules logiques.

Nous postulons que deux objets distincts du monde peuvent dénoter deux concepts distincts mais deux concepts distincts ne dénotent pas nécessairement deux objets différents.

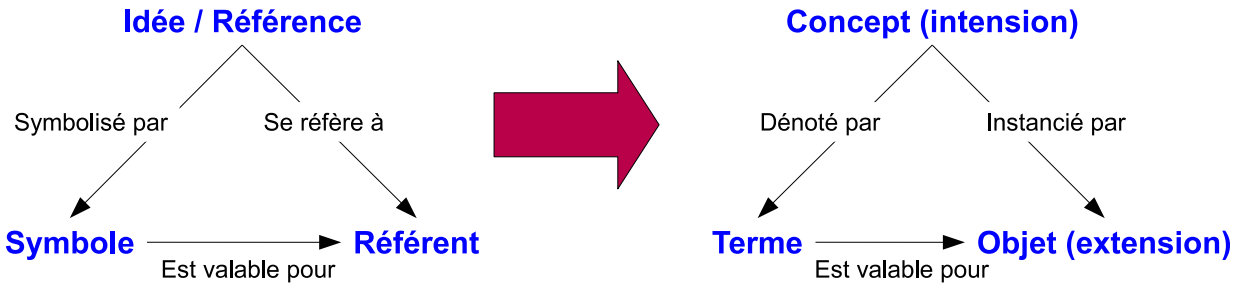


FIGURE 1.1 – Le triangle sémiotique et son interprétation ontologique.

1.1.1 Ontologie ou ontologies

Le mot ontologie est une création du siècle des lumières (circa 1721). Etymologiquement parlant, il est le résultat de la concaténation du mot grec "on" à partir du participe présent substantivé du verbe être "einai" et des mots "logia" (théorie) et "logos" (parole). L'ontologie est "science de l'être en tant qu'être" [Christophe Roche].

Dans [105], l'auteur montre que dans *"Der Streit um die Existenz der Welt"* R. Ingarden distingue trois types d'ontologies :

1. L'Ontologie existentielle (avec un grand O), branche de la philosophie. Elle cherche à définir les différents modes d'existence des objets.

2. L'ontologie matérielle qui étudie les aspects qualitatifs des objets. Elle se référe à leur existence générale (matérielle, informationnelle, sociale).

3. Enfin l'ontologie formelle qui explore la forme des objets. Elle est généralement liée à une logique dont le rôle est de spécifier les différentes propriétés des objets par la théorie de la quantification et par la théorie des modèles.

Une ontologie peut être appréciée de deux manières différentes, soit comme une classification, soit comme une catégorisation. Pour reprendre l'explication donnée par J.-P. Desclés : "Une classification ne représente qu'une simple distribution des objets dans des classes, tandis qu'une catégorisation contient, en plus des classes, les relations entre ces classes. Une catégorie est plus qu'une classe. Elle ne se définit pas par elle-même et rentre dans des hiérarchies." [111]

1.1.2 La notion de concept

La représentation de la connaissance peut être matérialisée par des langages issus de l'intelligence artificielle comme les langages-objets, à classes ou à prototypes et dont la définition correspond aux attributs communs à leurs instances. Ces objets sont dans une relation hiérarchique de spécialisation. Ce ne sont pas des représentations des connaissances, mais simplement des outils pour la représentation d'entités manipulables. D'autres formalismes reposent sur la logique comme le Calcul des Situations qui s'appuie sur la logique du premier ordre. Mais en se référant à Roche dans [118], la logique n'est plus épistémologique, et cela pose un problème.
En effet, d'un point de vue épistémologique, la notion de concept n'est pas une formule bien formée de la logique. Le concept est une primitive logique dont la preuve est son existence (comme pour les logiques constructives) et non une notion bivalente de la vérité (comme pour la logique classique).

Il existe une distinction entre les concepts théoriques - propriétés non observables - et les concepts observables de la réalité physique.

Généralement, les concepts théoriques posent un problème quant à leurs interprétations sémantiques. En effet, le sens d'un concept peut être défini de deux manières différentes, soit en extension, soit en intension. L'intension est elle-même composée de deux parties :

- L'ensemble des relations logiques entre les concepts (les propositions).
- L'interprétation de la théorie, qui nécessite la définition des règles d'interprétation pour établir un lien entre les concepts théoriques et les concepts observables.

Á la lumière de la logique, la notion de concept peut être interprétée de deux manières :

- L'une, une vue locale, qui considère le concept comme une fonction propositionnelle avec pour argument un objet et pour valeur de vérité l'expression de l'appartenance de l'objet au contexte du concept [111].
- L'autre, une vue hiérarchique, qui utilise les relations méréologiques, c'est-à-dire la définition d'un tout et de ses parties, et qui conçoit un concept comme un fragment d'un concept plus général [102].

1.1.3 Les relations ontologiques

L'intelligence humaine permet d'établir des rapports entre les concepts appelés relations. Ces rapports dépendent du degré d'abstraction, de la nature des concepts, etc.

De ces rapports se sont dégagés deux grands types de relations : les relations logiques et les relations ontologiques. Elles induisent le plus souvent notamment des rapports d'ordre hiérarchique entre les concepts.

Les relations logiques représentent des rapports d'abstraction entre les concepts. Ces rapports peuvent être l'inclusion, l'exclusion, etc. Par exemple : l'humain est un animal rationnel, ce dernier étant un être vivant.

Les relations ontologiques déterminent des rapports entre concepts dont les objets référents sont en relation de présence ou de contiguïté. Par exemple : la roue est une partie de la voiture. Elles traduisent souvent, dans un cadre terminologique, les relations partitives (hiérarchiques) et les relations associatives (le plus souvent non hiérarchiques). Les relations partitives sont celles de la méréologie [126]. Les relations associatives existent entre les concepts, soit par vertu (e .g. cause et effets), soit par l'usage. Les deux relations utilisées en terminologie sont les relations séquentielles et les relations topologiques.

Il n'est pas toujours possible d'identifier et d'expliquer l'abstraction des relations entre les concepts. Et cette différence fondamentale entre la relation logique et la relation ontologique est nécessaire pour la représentation du réel. Par exemple, la relation espèce genre (Paul, humain) relève fondamentalement d'une représentation effectuée par la pensée, alors qu'une relation méréologique de tout et partie est formée à partir de la connaissance de la nature et de la nature des objets du monde.

Voici quelques relations utilisées pour la représentation des connaissances dans les systèmes d'informations :

1. Relation d'Identité : relation sémantique qui existe entre deux concepts qui ont la même syntaxe, les mêmes attributs et les mêmes opérations.
2. Relation de Synonymie : relation sémantique qui existe entre deux noms qui expriment le même sens.
3. Relation de Classification *sorte de*, entre deux concepts exprimant que l'un est un cas particulier de l'autre.
4. Relation d'Homonymie : un même nom peut avoir deux sens différents.
5. Relation d'Antonomie : relation entre deux concepts totalement disjoints.

1.1.4 L'ontologie formelle

La première définition de l'ontologie formelle est donnée par le Philosophe Edmund Husserl « Prolégomènes à la logique pure » (Logische untersuchungen) (1900/01). Dans cet ouvrage, l'auteur dessine les prémisses d'une distinction entre la logique formelle et l'ontologie formelle. Pour lui les logiques formelles concernent les relations d'inférences, les notions de consistance et de vérité et les connexions entre des valeurs de vérité, alors que les ontologies formelles concernent

les connexions entre les choses, les objets et leurs propriétés, les relations de tout et parties et les collections. Elles manipulent des structures et des relations qui exemplifient des objets de domaines de la réalité. La base de l'ontologie formelle de Husserl repose sur la méréologie (il existe deux définitions de la méréologie, l'une de Husserl, l'autre de Lesniewski) [102], la théorie des dépendances et la topologie.

La notion de dépendance permet de mettre en évidence que les qualités de l'objet qualifié, le sont, là où les parties extensionnelles peuvent être appréhendées comme des objets en soi, indépendamment du tout. Cette distinction est importante pour parler des propriétés d'un objet. Il existe d'autres propriétés qui, elles, relèvent de la forme (contour) de l'objet, de sa constitution (matière), etc. Elles sont également attribuées à l'objet tout entier, les dimensions qualitatives ne pouvant pas être pensées indépendamment de l'objet qu'elles qualifient. Les dépendances factuelles (extensionnelles) d'un objet sont indépendantes du tout auquel elles appartiennent. Le tout dépend alors de la partie (un ascenseur dépend de ses composantes) qui peut être totalement indépendante.

Les ontologies formelles de Husserl peuvent se révéler comme des outils précieux pour la modélisation de la polysémie logique, puisqu'il est relativement aisé de traduire la théorie des catégories Husserliennes dans le langage de la théorie des types.

Barry Smith dans [127] présente une étude moderne de l'ontologie formelle dont les fondements méréologiques reposent sur la théorie de Lesniewski. Il définit dans [128] les ontologies comme étant une famille d'arbres. Chaque arbre est le reflet d'une vue spécifique dans un domaine considéré (différent niveau de granularité) comme par exemple microscopique, mésoscopique ou macroscopique.

Guarino [71] a proposé de décrire des méta propriétés des ontologies formelles avec une logique appropriée. Il définit par exemple des critères tels que l'unité, l'identité, la dépendance...

1.1.5 Les ontologies de domaine

Un système de relations logiques peut être considéré comme le modèle d'une ontologie de domaine si à chacune de ses relations correspond une interprétation validée par la communauté du domaine concerné. Le système complet est une représentation explicite d'une conceptualisation du domaine comprise comme un ensemble à la fois de situations et de connaissances du domaine. Une ontologie de domaine spécialise les termes d'une ontologie formelle qui sont appelés termes pour la description d'une situation.

1.2 La formalisation des ontologies

Il existe plusieurs manières d'interpréter le mot ontologie pour définir les ontologies formelles et Nicolas Guarino définit les différentes interprétations suivantes :

i L'ontologie est un système conceptuel informel.

ii L'ontologie est une sémantique formelle.

iii L'ontologie est une spécification d'une conceptualisation.

iv L'ontologie est une représentation conceptuelle d'un système par une théorie logique.

v L'ontologie est un vocabulaire utilisé par une théorie logique.

vi L'ontologie est un niveau méta permettant de spécifier une théorie logique.

Les interprétations *i*, *ii* et *iii* conçoivent les ontologies comme une sémantique conceptuelle, qu'elles soient formelles ou informelles. Les interprétations *iv*, *v* et *vi* représentent l'ontologie comme un objet syntaxique. L'interprétation *iv*, est la définition qui est actuellement la plus souvent retenue comme définition de l'ontologie en intelligence artificielle. Elle est classée comme syntaxique. Avec l'interprétation *iv*, l'ontologie n'est rien d'autre qu'une théorie logique. Avec l'interprétation *v*, l'ontologie elle-même n'est pas une théorie logique mais uniquement le vocabulaire utilisé par cette théorie. Enfin, avec l'interprétation *vi*, une ontologie est une spécification au niveau méta d'une théorie logique, dans le sens où elle spécifie l'architecture des composants utilisés par une théorie particulière des domaines.

Pour notre étude, les ontologies nous permettront de disposer d'un vocabulaire pour la définition des contextes (point v). Nous nous appuierons donc ainsi sur des conceptualisations existantes formant un consensus pour un domaine d'application.

1.2.1 L'épistémologie pour la représentation des connaissances

L'épistémologie peut se définir comme une discipline de la philosophie qui étudie la nature et les sources de la connaissance. D'un point de vue logique, la connaissance est identifiée par les propositions dont la structure formelle permet de déduire de nouvelles connaissances. A la différence des théories logiques qui manipulent des prédicats abstraits, le niveau épistémologique introduit la notion de concept comme une structure primitive de la connaissance (e.g., le Contexte). En épistémologie, les définitions de ces concepts sont données par des formules. Ainsi en physique, le concept de force est défini par $F = m\gamma$ (avec m la masse et γ l'accélération) et cette définition est démontrée. L'épistémologie, dans un sens strict est une étude portant sur le fondement de la connaissance des sujets cognitifs, i.e., comment sont représentées les preuves des phénomènes de la réalité. Cela peut être par exemple comment sont instanciées des classes.

1.2.2 Les contraintes sémantiques ("Ontological commitment")

Dans la communauté de l'IA, la notion de contrainte sémantique fut introduite par Grüber comme la spécification d'un vocabulaire commun. Cette spécification est un ensemble de terminologies, d'axiomes et de contraintes sémantiques consistantes avec ces axiomes. Les contraintes sémantiques sont une correspondance entre un langage logique et un ensemble de structures sémantiques. Ces contraintes sémantiques s'expriment à l'aide d'une logique modale dotée de primitives méréologiques et topologiques.

La classification de la représentation des connaissances formelles par rapport aux différentes primitives utilisées par Guarino est présentée à la figure 1.2 et dans le tableau 1.1

La couche logique constitue la primitive de base. Elle donne une sémantique formelle des relations entre les objets du domaine. Il n'existe pas d'hypothèse sur la nature de ces relations, qui sont totalement générales. Ce niveau est appelé le niveau "formalisation" et l'interprétation donnée y est totalement arbitraire.

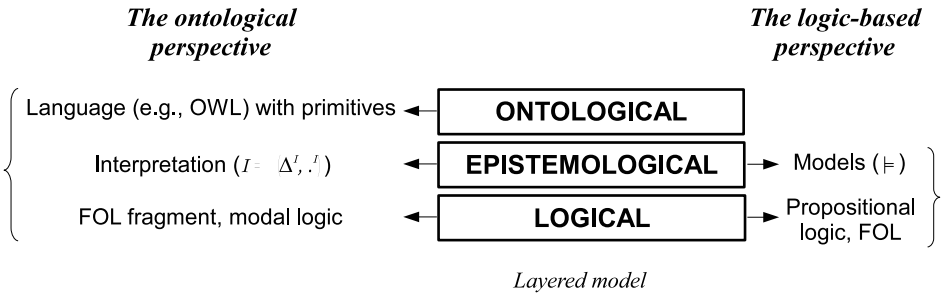


FIGURE 1.2 – Perspective traditionnelle de modélisation de la connaissance.

Niveaux	Primitives	Interprétation	Caractéristiques
Logique (premier ordre)	Prédicats, fonctions	Arbitraire	Formalisation
Epistémologie	Structures (relations)	Arbitraire	Structure
Connaissance	Connaissances (relations)	Contraintes	Sens
Conceptuelle	Conceptuelles (relations)	Subjective	Conceptualisation

TABLE 1.1 – Les différents niveaux d'abstraction selon Guarino

Le niveau épistémologique (introduit par Brachman comme lien entre le niveau logique et le niveau conceptuel) permet de spécifier une structure formelle d'unité conceptuelle et ses relations comme des unités conceptuelles (indépendamment de leurs connaissances internes)[19].

Ainsi, au niveau épistémologique, la notion de concept générique est introduite comme une primitive. Les concepts qui correspondent aux prédicats unaires de la logique possèdent une structure interne assemblée pour former des concepts ou des relations binaires (rôles). Le niveau épistémologique est appelé le niveau des structures.

Toute structure logique L, décrite dans la couche épistémologique, est construite sur une structure S qui décrit une théorie T. En d'autres mots, T spécifie quelle contrainte ontologique elle est supposée satisfaire dans la couche épistémologique [60].

Au niveau connaissance, les contraintes sémantiques associées aux primitives du langage sont spécifiées explicitement. Ce niveau est appelé niveau du sens. Bien sûr, cette caractérisation est approximative.

Au niveau conceptuel, les interprétations des primitives sont définies d'une manière cognitive indépendamment des concepts comme les actions et les rôles.

1.2.3 La conceptualisation

Pour Grüber [69], une ontologie est une spécification explicite d'une conceptualisation. Il sous-entend donc par explicite, que les objets sont décrits d'une façon extensionnelle. Pour Guarino au contraire, cette position n'est pas acceptable. Aussi, il propose de représenter les ontologies par une structure intensionnelle s'inspirant de la sémantique de Montague [103]. Il représente une

conceptualisation par la structure intensionnelle : $\langle W, D, R \rangle$ où W est un ensemble de mondes possibles, D des objets du domaine et R un ensemble de relations intensionnelles de D.

La conceptualisation peut être spécifiée en respectant un certain nombre de règles telles que :

- L'ingénierie des ontologies est une branche de l'ingénierie et des connaissances qui utilise l'Ontologie pour construire des ontologies.
- Les ontologies sont une sorte de base de connaissances.
- Toutes les ontologies ont une conceptualisation.
- La même conceptualisation peut sous-entendre des ontologies différentes.
- Deux bases de connaissances différentes peuvent correspondre à une même ontologie.

1.3 La représentation des ontologies

Pour décrire une ontologie particulière, il est de coutume d'utiliser une taxonomie de base des entités du monde à représenter (i.e., que les entités sont organisées suivant une relation d'ordre partiel de généralisation et de spécialisation). D'un point de vue mathématique, une taxonomie est un treillis [3] dont les noeuds sont des classes ontologiques.

Le mot classe est couramment utilisé comme référence pour parler des entités d'une ontologie, entités qui désignent des concepts ou des catégories présents dans le monde devant être modélisé. Dans une taxonomie, les classes sont données dans une relation d'ordre partiel appelée subsomption . La subsomption - is-a - ou hyponymie, est la relation sémantique entre deux concepts selon laquelle l'extension du premier est incluse dans l'extension du second. Elle est à opposer à la relation d'hyperonymie - is-a-type-of - qui représente la hiérarchie pouvant exister entre un lexème et un autre selon laquelle l'extension du premier terme, plus général, englobe l'extension du second, plus spécifique. Par exemple, il est possible de dire qu'"animal" est un hyperonyme de "chien" et que "Mabrouk" est un hyponyme de "chien", qui est lui-même hyponyme d'"animal".

N.B. : Une taxonomie ne contient pas d'instance de classe.

Les approches de représentations des ontologies reposent souvent sur la théorie des ensembles. Les opérations de base en sont l'union, l'intersection, le sous-ensemble, etc.

Par contraste avec la théorie des ensembles, la méréologie concerne les relations entre les parties et un tout. Elle cherche à identifier les parties d'une entité particulière et comment ces parties sont liées à d'autres. D'un point de vue historique, c'est Husserl qui fut le premier à utiliser une sorte de méréologie comme point de départ pour les ontologies. L'avantage de la méréologie par rapport à la théorie des ensembles est qu'un domaine peut être modélisé sans connaître exactement les parties qui le composent.

Les ontologies sont également une méthodologie et un ensemble de principes qui permettent de capturer une relation formelle, ce qui touche de nombreux domaines dans le monde actuellement, puisqu'elles permettent la réutilisation et le partage des connaissances. Elles sont ainsi un

3. Une relation d'ordre partiel R réflexive, transitive et antisymétrique est un treillis ssi pour tout couple X et$Y$ en relation par R, $sup(X, Y)$ (majorant) et $inf(X, Y)$ (minorant) existent.

consensus, une norme commune à plusieurs disciplines. Ceci se retrouve dans les travaux actuels sur le Web sémantique.

1.4 La conception des ontologies

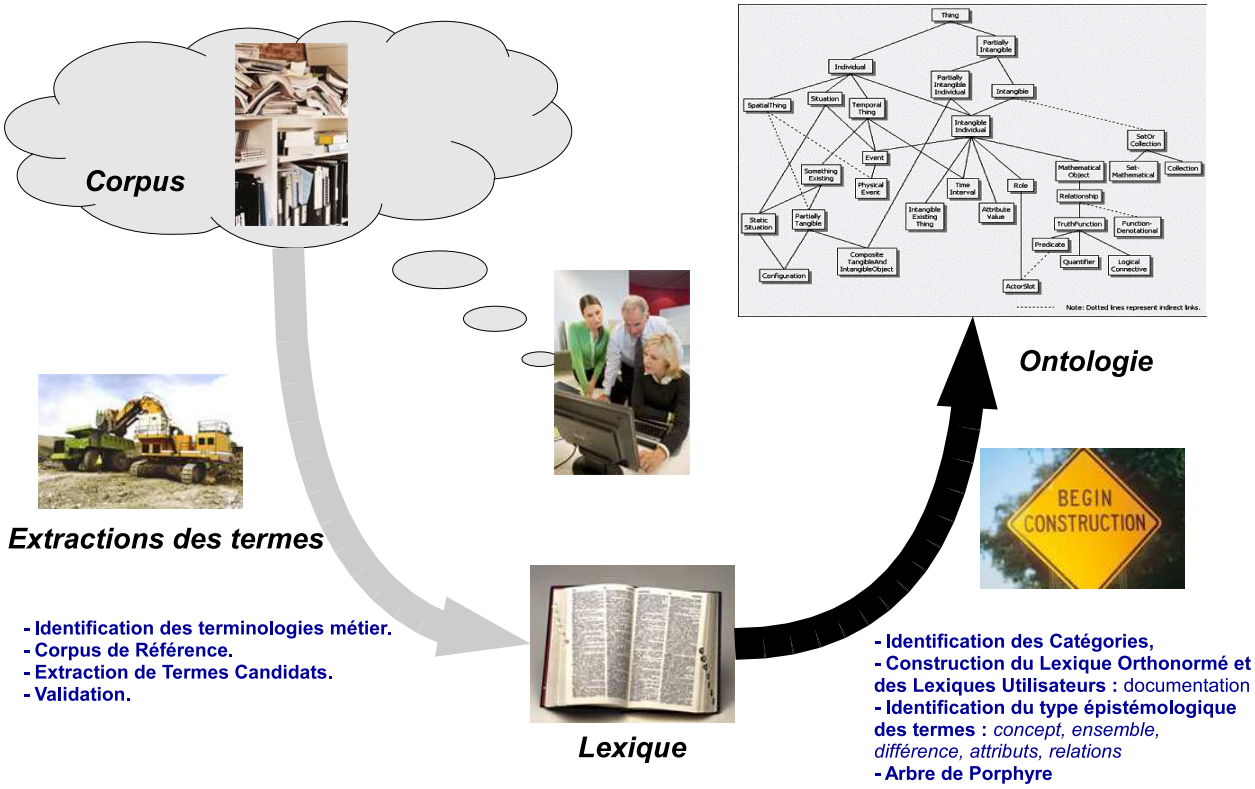


FIGURE 1.3 – Cycle de vie en spirale de l'ontologie OK

Pour mieux comprendre les ontologies, un bon moyen peut être d'étudier les principes de la conception d'une ontologie, en partant par exemple du cycle de vie en spirale du modèle ontological knowledge décrit à la figure 1.3.

Selon Christophe Roche la définition d'une ontologie peut s'énoncer ainsi :

"L'ontologie est une conceptualisation d'un domaine à laquelle sont associés un ou plusieurs vocabulaires de termes. Les concepts se structurent en un système et participent à la signification des termes. L'ontologie est définie pour un objectif donné et exprime un point de vue partagé par une communauté. Une ontologie s'exprime dans un langage et repose sur une théorie (sémantique)

garante des propriétés de l'ontologie en termes de consensus, cohérence, réutilisation et partage."

Partant de cette formulation, l'exemple de la figure 1.3 se décompose en deux étapes :

La première étape consiste en une étude, qui se fait en collaboration avec des spécialistes du domaine (e.g. économistes, ingénieurs) d'un corpus d'informations. Elle se développe en quatre points :

1. Identifier la terminologie du domaine étudié.
2. Etablir un corpus de référence.
3. Extraire les termes retenus.
4. Valider le résultat.

La seconde étape est la réalisation de l'ontologie. Elle se décompose également en quatre points :

- Le premier point est l'identification des catégories qui correspondent à des prédicats unaires déterminés par une conjonction de prédicats de plus haut niveau :

$$Contexte(x) \triangleq Contrainte(x) \wedge Propriété(x)$$

- Le second point est un travail de construction d'un lexique orthonormé (les termes qui le composent dénotent un concept) et des lexiques utilisateurs (synonymes) à partir de la documentation fournie par les spécialistes.

- Le troisième point cherche à identifier des types épistémologiques (l'essence des termes et définitions des concepts par l'expression de leurs différences spécifiques, des termes étant par exemple les concepts, les ensembles, les différences, les attributs et les relations).

- Le quatrième point est la construction d'un dit "Arbre de Porphyre" (e.g. fig. 1.4). Une définition de cet arbre pourrait s'énoncer ainsi :

  Porphyre dans "l'Isagore" fournissait une représentation hiérarchique des êtres permettant de structurer chacune des dix catégories distinguées par Aristote. La formulation de cette hiérarchie présentée par Porphyre, sans mention du jeu des différences, sur le seul exemple de la catégorie de substance, première des dix catégories, a donné lieu à la représentation de ce qui est plus communément appelé une "Arbre de Porphyre"

Un exemple d'ontologie est présenté à la figure 1.5

1.5 Les langages de représentation des ontologies

Il existe un grand nombre de langages pour la représentation des ontologies. Ces langages résultent d'un compromis entre trois propriétés :

1. La lisibilité (readability), i.e. comment décrire les objets du Monde.

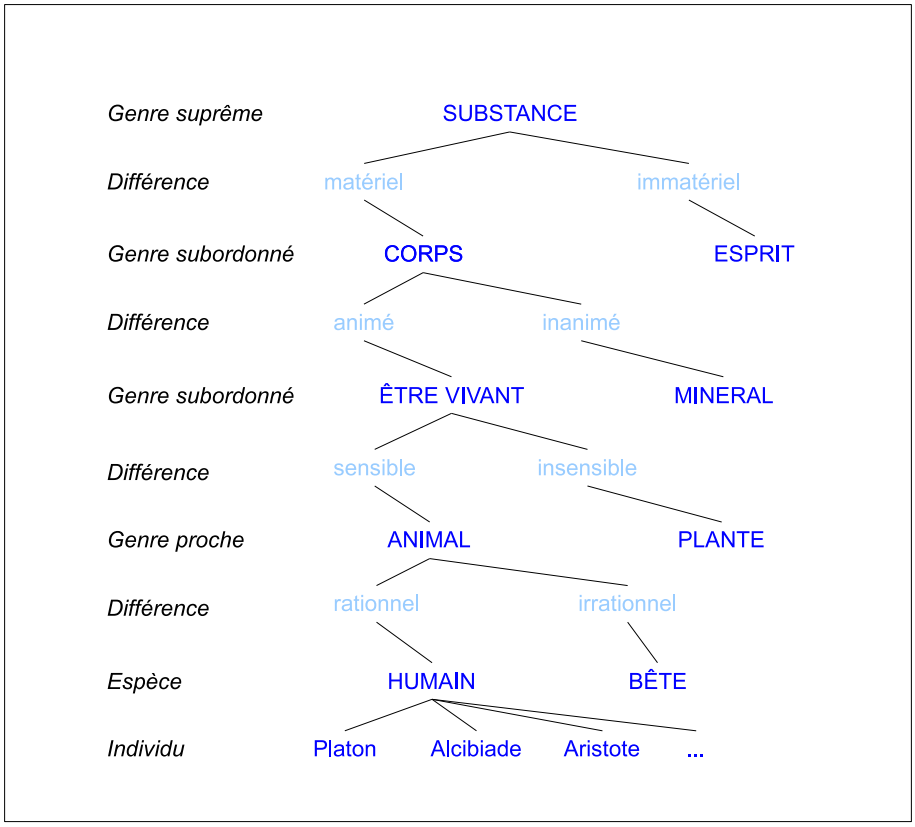


FIGURE 1.4 – Arbre de Porphyre

2. L'expressivité, i.e. ce qui peut être décrit.
3. L'inférence, i.e. ce qui peut être déduit des informations présentes dans l'ontologie.

Ces langages peuvent être classés en deux familles, ceux qui reposent sur la logique et ceux qui reposent sur les cadres (frame) de Minsky [101].

Les langages logiques utilisent le plus souvent la logique du premier ordre ou les logiques de description, [107], dont OWL [50] en est un exemple. Ces langages garantissent une syntaxe et une sémantique clairement définies et un langage d'échange. Le problème est d'ordre épistémologique et linguistique. En effet, si la logique permet de garantir la justesse d'une base de connaissances, ceci est cependant nécessaire a posteriori et non a priori.

Les langages de cadre (frame), définissent les primitives les plus utilisées pour la définition des ontologies, i.e. les classes (concepts), les attributs (propriétés) et une relation d'héritage entre les classes formant une taxinomie. Le langage de cadre le plus connu se nomme ONTOLINGUA [69]. Le problème avec les langages de cadre est qu'ils ne sont qu'une technique et non pas une théorie formelle de la connaissance. La relation de subsomption est plus qu'une simple relation d'héritage [4] entre des classes, et les propriétés sont plus que de simples attributs.

4. Il existe des cas où certaines classes ont en commun des variables, méthodes de traitement ou même des signatures de méthode (i.e., que l'héritage concerne l'état, le comportement et les contraintes). Le mécanisme d'héritage entre des classes permet de factoriser ces membres communs. Le principe de l'héritage est de créer

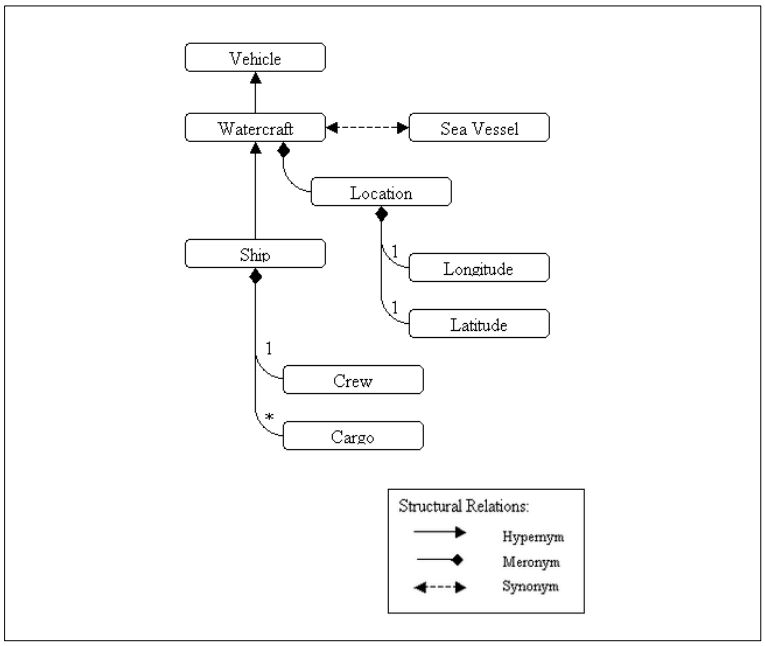


FIGURE 1.5 – Exemple d'ontologie

Les ontologies pour l'ingénierie des connaissances se distinguent principalement des bases de connaissances de l'intelligence artificielle des années 80 sur trois points :

- La recherche de propriété logique.
- La volonté de partage.
- La recherche d'un format lisible.

Les ontologies en ingénierie des connaissances n'ont pas pour objectif de comprendre le monde mais d'en représenter les objets à des fins de manipulation informatique. Ce sont des " langages de représentation ".

Elles relèvent d'un programme logiciste dans la construction des connaissances sur une base logique (sémantique) et d'un programme mécaniste (syntaxique) dans la représentation d'entités manipulables (Christophe Roche). La syntaxe et la sémantique forment donc une théorie pour la représentation des connaissances. Ces langages (e.g., OWL) permettent un raisonnement limité sur les ontologies. Ils sont souvent associés à de la programmation logique pour introduire des variables et pour pouvoir raisonner avec des règles [15]. Cependant, le raisonnement au niveau

une relation, dite d'héritage, entre deux classes pour définir qu'une classe est une spécialisation de l'autre. On exprime ce fait en disant que : la classe la plus spécifique hérite de la classe la plus générale. L'idée sous-jacente à la définition de l'héritage est de pouvoir organiser les classes de manière hiérarchique (une relation d'héritage se lit : "est un" ou "est une sorte de").

méta (i.e., sur les propriétés des relations ontologiques) reste un challenge actuel.

1.6 De l'utilisation des types dépendants pour la spécification des ontologies

En 1985, Hobbs [74] remarqua que les difficultés rencontrées pour définir la sémantique des langues naturelles étaient dues en partie à des problèmes rencontrés lors de la spécification de la nature même des relations existantes entre une langue et la réalité. Il constata en outre, qu'en modélisant une théorie du monde (e.g. une ontologie) basée sur le bon sens et la découverte plutôt qu'inventée, la spécification de la sémantique devenait alors triviale. Par ailleurs, dans [33] Cocchiarella proposa d'identifier la logique à un langage, contrastant ainsi avec la vision classique qui la voit comme un calcul. La logique possède alors un contenu ontologique, et pour cela l'auteur indique qu'il est nécessaire d'utiliser une théorie des types, plutôt que la théorie des ensembles, comme cadre de travail.

Dans [140] l'auteur discute des relations de proximité entre le langage et la connaissance et il explique que, pour comprendre le sens commun d'un langage, il est nécessaire d'utiliser une structure ontologique fortement typée. Ainsi, par isomorphisme entre les structures ontologiques et la logique interne de celles-ci, il est possible de raisonner sur le sens commun (figure 1.6).

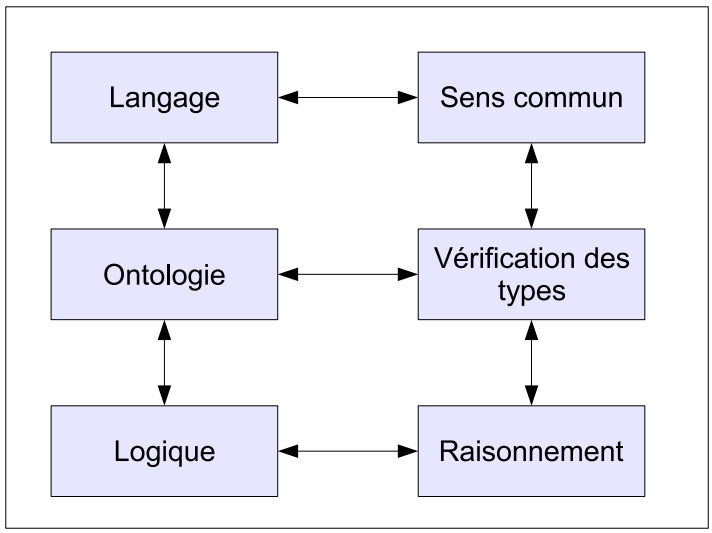


FIGURE 1.6 – Langage, logique et ontologie

Une ontologie de domaine est une spécification formelle des relations qui existent entre des concepts. Elle apparaît comme une outil approprié pour l'exploitation d'une théorie des types, puisqu'elle permet de décrire un vocabulaire et son interprétation pour un domaine du discours particulier. Il est possible de distinguer dans les ontologies la notion de type et la notion d'objet. Les types ou " expressions syntaxiques " représentent les concepts ou symboles de classe, les noms de relation, les expressions logiques, etc. alors que les objets sont des preuves de l'existence de ces types.

Représenter une ontologie par une théorie des types permet un gain d'expressivité par rapport aux solutions définies par la logique du premier ordre. En effet, avec celle-ci, il n'est pas possible :

- de différencier les individus et les classes d'individus,
- de parler des relations entre les individus.
- de spécifier des propriétés sur les relations, par exemple :

$$transitive(R) \triangleq \forall x\ \forall y\ \forall z\ (R(x,y) \wedge R(y,z) \Rightarrow R(x,z))$$

Alors qu'avec une représentation des ontologies par une théorie des types :

- Les individus sont des preuves pour des types (=classes),
- Les relations entre types sont prouvées par des relations (paires emboitées) entre individus.
- Les propriétés sur les relations peuvent être exprimées par des Σ-types.

1.7 Synthèse

Le besoin d'un niveau ontologique bien défini est nécessaire dans les applications portant sur la connaissance en IA. En effet, ces systèmes formels, en définissant des règles d'inférence pour mieux comprendre le monde, permettent de résoudre des problèmes difficiles de l'IA.Comme par exemple en planification où le rôle le plus évident des ontologies est de fournir une représentation et des structures qui ne changent pas. Cependant, l'ontologie ne se réduit pas à être un objet de description. Elle est également concernée par les événements, l'état et autres entités temporelles. Les ontologies offrent une standardisation(compréhension entre personnes et logiciels) du sens(pour lever les ambigüités) de chacun des termes (mot de la langue naturelle qui désigne un concept) qui sont employés dans un domaine de connaissance et une description des relations existantes entres les concepts représentés par les termes.

Il existe une relation entre un raisonnement formel et une représentation qui est semblable à celle pouvant exister entre l'épistémologie (comprise comme l'étude de la connaissance) et les ontologies formelles (étude de ce qui "est").

La nature ontologique des choses doit être comprise dans le cadre de ce mémoire par des catégories conceptuelles bien définies. Ce sont : le rôle (prédicat binaire), le concept et la propriété (prédicat unaire), avec des relations de "subsomption" et de "tout à partie" entre les concepts.

De ce point de vue, les ontologies permettent de représenter les connaissances structurellement invariantes de la réalité, circonscrites localement par des ontologies de domaine pour gagner en précision et limiter l'espace de recherche lors de la conception et de l'exécution.

Force est de constater que, dans le cas d'un environnement plus spécifique avec des contraintes idiosyncratiques de la conceptualisation partagée, des théories plus élaborées peuvent être prises en compte pour apporter une plus grande précision et la dynamique qui manquent aux ontologies. Ce sont les théories du contexte.

2

De la notion de contexte

FIGURE 2.1 – Les contextes possibles

Concevez toujours une chose en la considérant dans un contexte plus large - une chaise dans une pièce, une pièce dans une maison, une maison dans un quartier, un quartier dans une ville.

[Eliel Saarinen]

L'objectif de ce chapitre est de donner une définition du mot contexte qui soit suffisamment expressive pour permettre de représenter l'ensemble des applications de KR&R qu'il est possible de réaliser par exemple avec le calcul des situations, [100], [65] et d'une complexité suffisamment faible pour pouvoir être susceptible d'une implantation dans des systèmes tels que l'informatique diffuse et les systèmes sensibles au contexte (context-aware systems).

Pour cela, il est nécessaire d'étudier les spécifications existantes de la notion de contexte en Intelligence Artificielle (IA), puis d'en tirer des contraintes en adéquation avec notre besoin de généralité et de précision, afin d'en déduire la définition recherchée.

Le second objectif sera de montrer quels modèles ou quelles théories sont les plus pertinentes pour satisfaire à la définition précédente.

2.1 Vers une définition du concept de contexte

La notion de contexte apparaît dans de nombreuses disciplines de l'Intelligence Artificielle. Par exemple dans la représentation des connaissances [86] ou le traitement des langues naturelles [122], le contexte est utilisé pour résoudre les ambiguïtés. Dans la recherche intelligente d'informations [66, 99] le contexte permet de raffiner les requêtes utilisateur et enfin dans l'interaction homme-machine [20] le contexte apporte une aide à la conception d'applications.

Les définitions du contexte ont été "largement" introduites et discutées dans la littérature - voir notamment [18] pour un panorama des propriétés du contexte - et dans ce mémoire, seuls seront résumés les points importants. Le mot contexte est souvent désigné pour décrire une multitude de choses allant des descriptions aux explications et à l'analyse. Sa signification est souvent implicite et laissée à l'interprétation du lecteur [1]. Les logiciels sont des systèmes représentationnels, et les définitions du contexte étant destinées à être traitées par un logiciel reflètent cet aspect.

Dans [110], l'auteur suggère une définition présentant le contexte comme "un concept subjectif défini par l'entité le percevant". Une telle définition est bien trop vague pour être utilisée dans la réalisation d'un système basé sur le contexte. Au contraire, d'autres auteurs tels que [133, 54] fournissent une définition du contexte bien trop spécifique d'un domaine ou d'une application pour pouvoir être réutilisée de manière systématique. Certaines définitions sont même parfois trop difficiles à utiliser en pratique [54].

Dans [124], les auteurs considèrent le contexte comme la localisation d'un utilisateur, sa situation sociale ainsi que les ressources qui l'entourent. De manière assez similaire, Schmidt et al. décrivent le contexte en utilisant un modèle comportant trois dimensions qui sont l'environnement (physique et social), l'état propre (état du composant, physiologique et cognitif) et l'activité (comportement et tâche en cours). [30] font une distinction entre les aspects actif et passif du contexte en spécifiant le contexte comme " un ensemble d'états et d'environnements qui déterminent le comportement d'une application dans lequel survient un évènement lié à une application qui concerne l'utilisateur ". Pour les auteurs, ces deux types de contexte contribuent à une meilleure compréhension dans les applications utilisant le contexte (context-aware). Ces définitions du contexte sont propres à certains types d'applications, de natures hétérogènes et ne sont pas des modèles bien spécifiés et partageables entre applications.

La définition la plus utilisée dans la modélisation du contexte est sans doute celle de [53] qui décrit le contexte comme " toute information pouvant être utilisée afin de caractériser la situation d'une entité. Une entité peut être une personne, un endroit, ou un objet considéré comme significatif dans l'interaction entre un utilisateur et une application, incluant également l'utilisateur et l'application ". Là encore, cette définition n'est pas suffisamment précise pour servir de base à une modélisation effective. Par ailleurs, d'autres auteurs [143] ont remarqué que

cette définition se limite à l'utilisateur et aux applications sans considérer la notion de réseaux afin d'étendre le contexte aux applications centrées-réseaux.

Dans [66] un contexte est défini comme une description des aspects d'une situation et l'auteur insiste sur la nécessité d'introduire une structure de contexte commune pour décrire les contextes des utilisateurs. Nous retrouvons cet aspect de la ré-utilisabilité dans [81] où les auteurs argumentent pour l'utilisation d'un format standard et simple pour modéliser les aspects de l'activité humaine qui sont associés aux éléments pertinents du contexte. Brézillon et Cavalcanti [23] ont établi une liste de définitions du contexte parmi lesquelles : "un ensemble de préférences et/ou de croyances", " un ensemble infini et partiellement connu d'hypothèses", "le résultat d'une interprétation ", " des chemins de recherche en recherche d'informations ".

Bien qu'il n'y ait pas vraiment de consensus autour de la notion de contexte, la spécification de celui-ci apparaît comme intimement liée à la notion de situation. Un contexte peut être vu comme l'ensemble des informations se rapportant à une situation particulière [131, 56, 29, 115, 88, 106]. En effet, une situation représente un ensemble de faits physiques - localisation spatiale, temporelle - ou fonctionnels, par exemple une tâche en cours de réalisation.

Dans [65] où l'auteur décrit les contextes comme un ensemble partiellement ordonné selon le niveau d'abstraction considéré.
Selon Brezillon [20] le contexte est un ensemble de connaissances. Le contexte est alors relatif à une étape donnée de la résolution d'un problème, et les connaissances sont organisées autour de cette étape. Dans le cadre des activités humaines [57], l'auteur insiste sur le lien existant entre contexte et activité tout en indiquant qu'ils sont séparés. Il conclut en précisant que le contexte est dynamique, redéfini au cours d'une activité divisée en une séquence d'actions, et que c'est dans ce contexte changeant que chaque action prend toute sa signification. Cette notion d'environnement changeant est également soulignée dans l'article de [38] où le contexte est vu comme une partie d'un processus plutôt que comme un état. Les auteurs ajoutent qu'un cadre conceptuel incluant des ontologies est nécessaire pour minimiser les erreurs d'interprétation au cours de l'évolution du contexte. Enfin, il apparaît que le contexte devrait, pour la plus grande part, être déduit automatiquement (Floréen06).

Bien que chacune de ces définitions mette en avant tel ou tel aspect du contexte, il n'y a pas de définition communément admise et chaque domaine ou système d'application sensible au contexte semble se trouver dans la position de constituer sa propre vision du contexte, répondant aux particularités d'une situation donnée. À partir de cet ensemble non exhaustif de définitions, plusieurs contraintes, qui satisfont à la fois les systèmes dépendant du contexte et les systèmes de raisonnement sur le contexte comme ceux présents en IA, peuvent être isolées :

1. Le contexte doit contenir des connaissances (ressources, faits), lesquelles sont dynamiques.
2. Le contexte est lié à un concept de nature intentionnelle et dynamique (e.g., une action, un processus, ...).
3. Il y a place pour plusieurs contextes dans une situation donnée,
4. la définition d'un contexte ne doit donc pas dépendre d'une application particulière.
5. Il doit être possible le cas échéant de réutiliser le contexte d'une manière dynamique dans de nouvelles applications.
6. Le contexte doit être calculable.

En partant de la définition suivante (dictionnaire) " Le contexte est l'ensemble des circonstances (donnant le sens) dans lesquelles s'insère un fait (situation, environnement)" [Le Robert] et des contraintes ci-dessus, il est possible de proposer la définition suivante :

Un contexte est l'ensemble des connaissances valides à un instant donné qui particularise une action et lui donne une signification particulière. Cet ensemble de connaissances est inclus dans l'ensemble des connaissances défini par la situation dans laquelle l'action peut s'exécuter.

Il est facile de vérifier que cette définition peut être l'objet d'un consensus vis-à-vis des définitions ci-dessus. La réutilisation du contexte est rendue possible via la notion de type de contexte. La dynamicité est assurée par le fait que, dans une situation donnée, n'existent que des objets (instances) appartenant à des types de contexte.

2.2 Vers une modélisation du contexte

2.2.1 Aspect *représentation* du contexte

L'aspect représentation du contexte s'intéresse au développement de formalismes reposant sur le contexte des connaissances et le raisonnement. Ces formalismes offrent une possibilité d'apprentissage (évolution dynamique) et fournissent une représentation simple et aisément lisible par un utilisateur de base, du modèle et de la compréhension de son évolution. Ici, l'utilisateur est le point central.

Les graphes contextuels (GxC) sont un exemple d'une telle approche. Ce formalisme fut développé à l'origine pour traiter un problème de résolution des incidents dans le métro [5]. Un GxC fournit une représentation qui repose sur la notion de contexte d'un problème par la donnée des processus opérationnels en tenant compte de l'environnement de travail. Au départ, il est un squelette des procédures répertoriées puis, progressivement, lorsque surviennent des incidents (éventuellement) il est enrichi par un opérateur qui applique les procédures dans un contexte donné.

Un graphe contextuel [21] est un graphe orienté acyclique avec un unique point d'entrée et un unique point de sortie. Les noeuds du graphe représentent des actions simples ou parallèles, des noeuds contextuels, des noeuds de recombinaison ou bien encore des activités.

Un chemin du graphe contextuel correspond à la solution d'un problème dans un contexte donné.

- Les actions sont des méthodes exécutables alors que les activités représentent un assemblage complexe d'actions.
- Les éléments contextuels sont les noeuds contextuels et les noeuds de recombinaison. Un noeud contextuel correspond à une instance d'un élément (e.g. Paul) alors que le noeud de recombinaison correspond à l'abandon d'une instance, suite à une action par exemple.
- Les sous-graphes sont la représentation d'un raisonnement local (diagnostic ou action) qui correspond à un but intermédiaire. Il est lui même un GxC.

5. (www.lip6.fr :SART)

– Les actions parallèles représentent (comme leurs noms l'indiquent) des actions pouvant être exécutées en parallèle.

La figure 2.2 donne un exemple de graphe contextuel. Les A_i représentent les actions, C_j, k une instance k d'un noeud contextuel $C_j(1, n)$ et R_l les noeuds de recombinaison.

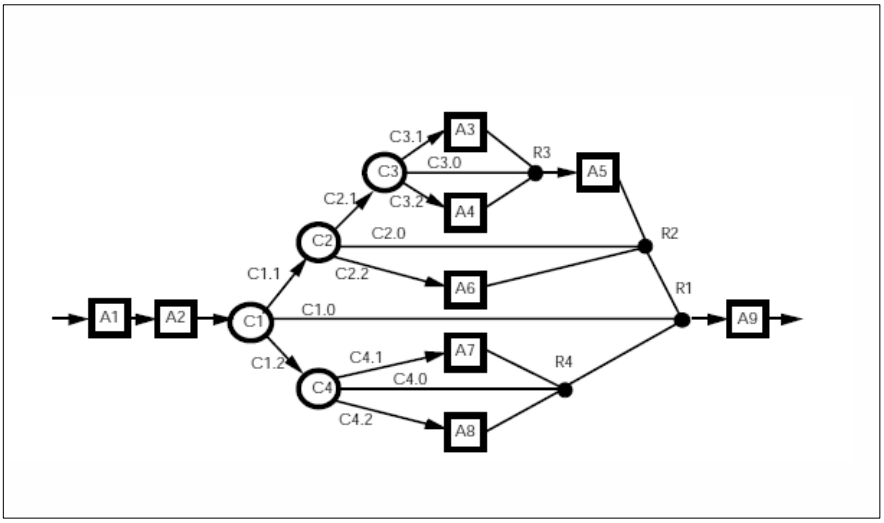


FIGURE 2.2 – Exemple de graphe contextuel

2.2.2 Aspect *modélisation* du contexte

Cet aspect est plutôt orienté vers l'emploi pratique et immédiat de la notion de contexte. La téléphonie mobile est un bon exemple parmi les applications mobiles et sensibles au contexte qui se sont beaucoup développées ces dernières années. Cette approche est dite centrée matériel. Le contexte physique prend maintenant une grande importance.

Il existe de nombreuses définitions pour ce type de contexte. Elles sont souvent liées à une application ou à un domaine particulier. Evidemment, ce type de définition rend difficile une extension à d'autres emplois.

Euzenat propose dans [58] un modèle pour l'informatique diffuse repris à la figure 2.3. Il présente une manière de constituer progressivement les informations de contexte (des capteurs à l'application). Dans ce modèle, les informations en provenance des capteurs sont réunies et combinées pour obtenir le modèle de contexte utilisé.

Les références relatives à la modélisation du contexte dans la littérature de ces quinze dernières années montrent que les rares modèles formels de contexte proposés reposent principalement sur deux approches différentes. L'une est centrée sur les ontologies, alors que l'autre tente de fournir une base logique à la notion de contexte.

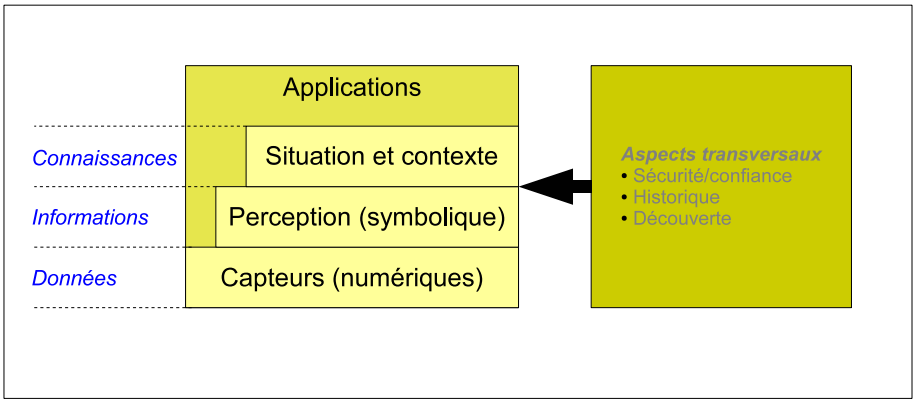


FIGURE 2.3 – Modèle de contexte en informatique diffuse

Les méthodes orientées ontologie constituent un cadre privilégié pour la spécification des concepts et de leurs relations [139, 70]. En effet, elles sont plus particulièrement adaptées à la modélisation des informations relatives aux aspects de la vie courante, offrant ainsi des structures de données manipulables par une machine. Le développement de modèles de contextes basés sur les ontologies permet un meilleur partage de la connaissance et garantit une réutilisabilité accrue. Ainsi dans [142], les auteurs décrivent une ontologie formelle pour capturer les propriétés générales d'entités contextuelles primitives et une collection d'ontologies de domaine avec leurs propriétés et ce, dans chaque sous-domaine. Le système CoBrA de [31] lui, fournit un ensemble de concepts ontologiques qui permet de caractériser des entités telles que des personnes, des localisations ou tous autres types d'objets dans leurs contextes. Quant à l'architecture d'agent avec superviseur (broker-centric), elle offre un support d'exécution pour les systèmes sensibles au contexte. Un état de l'art récent sur les approches les plus significatives dans la modélisation du contexte [130] établit que les approches les plus prometteuses sont celles qui sont basées sur les ontologies.

Pour les modèles utilisant la logique, l'information contextuelle est introduite, mise à jour et supprimée sous forme de faits ou inférée à partir d'un ensemble de règles. Au début des années 90, les bases formelles les plus significatives pour le raisonnement à partir des contextes furent explorées à partir des logiques centrées sur le calcul des situations. Ainsi les travaux de [100] décrivent les contextes comme des objets de première classe, en introduisant par exemple le méta-prédicat $ist(c, p)$ pour affirmer que l'assertion p est vraie dans le contexte c. Des règles d'extension du contexte - les *lifting rules* - permettent de relier une valeur de vérité dans un contexte à une valeur dans un autre contexte. Ajouté à cela, des opérations telles qu'entrer et sortir d'un contexte sont introduites. Ces travaux servent de point de départ pour une formalisation en logique propositionnelle [25] qui est étendue par la suite aux langages de premier ordre [24].

L'approche privilégiée par Giunchiglia [61, 65], elle, repose sur deux hypothèses :

- le principe de localité où le raisonnement est dérivé à l'intérieur d'un contexte,
- le principe de compatibilité où des relations apparaissent entre des processus de raisonnement propres aux différents contextes.

L'auteur considère les contextes comme des micromondes qui possèdent chacun leur propre système de règles. Il existe une approche connue sous le nom de système multicontextes qui considère qu'un contexte est un sous-ensemble de l'état complet d'un système. Ces contextes sont alors vus comme des théories partielles interagissant entre elles au travers de règles d'association, lesquelles permettent d'inférer une conclusion dans un contexte à partir d'une hypothèse dans un autre.

Plus récemment, une approche des contextes explorant des idées issues de la théorie des types a été développée dans [137][138]. Elle manipule trois constructions syntaxiques primitives qui proviennent de la théorie des types à savoir : l'identité, l'application fonctionnelle et la λ-abstraction. Dérivées de travaux précédents relatifs à la logique intensionnelle [103], les propositions y sont définies comme des ensembles de mondes possibles, tandis qu'un contexte y est identifié par un ensemble de propositions. Dans l'étude des interactions personne-machine, le contexte est perçu comme une caractéristique de l'interaction - une propriété de l'information, ou des objets - dans laquelle les contextes et les activités sont intimement reliés - *context arises from the activity* - [57].

2.2.3 Les techniques pour la modélisation des contextes

La modélisation des contextes est en général le résultat d'une correspondance établie entre une situation du monde réel et des modèles de contextes utilisés par un système quelconque. Cette correspondance pose quelques problèmes. En effet, elle est incomplète puisque la représentation donnée par le système n'est pas capable de capturer l'ensemble des aspects du monde réel (e.g. problème du cadre, sémantique du monde ouvert). Elle est dite incertaine, à cause de l'utilisation de méthodes indirectes associées à des heuristiques, et imprécise à cause des méthodes employées. Enfin, elle est inconsistante, puisqu'elle est incertaine et imprécise. Ces différents problèmes sont tout particulièrement vrais pour des informations d'un contexte avec un haut niveau d'abstraction, contrairement aux informations fournies par des capteurs.

Selon [130], il est possible de donner six catégories de techniques pour la modélisation des contextes, qui sont :

1. La technique des paires de valeurs : cette technique repose sur la définition de collections de paires de valeurs (e.g. des tables d'associations).
2. La technique des Mark-up schemes : avec cette technique les informations sur les contextes sont fournies par un langage à balises (souvent XML), par exemple CSCP [79] qui est une technique pour la représentation des contextes utilisant RDFS.
3. Les techniques graphiques : elles utilisent des langages graphiques pour modéliser les contextes. Des exemples de telles techniques sont les diagrammes UML et les graphes contextuels.
4. Les techniques orientées objets : elles utilisent des outils comme l'encapsulation et l'héritage. Des exemples types de ces techniques sont CUES [125] et Active Object Model [32].
5. Les techniques logiques : les logiques sont utilisées pour pouvoir raisonner sur les propriétés du contexte. Les logiques utilisées pour formaliser le contexte sont souvent des logiques du

premier ordre, parfois des logiques spécialisées. Il existe également une logique qui est une extension du calcul des situations [100].

6. Les techniques utilisant les ontologies : ce sont les ontologies qui permettent de modéliser les informations sur le contexte. Elles profitent ainsi des possibilités de raisonnement permises par les ontologies.

Type d'approche	C	P	R	G	N	A
Paires de valeurs	-	-	-	-	-	+
Schémas de marquage	+	++	-	-	+	++
Approches graphiques	-	-	+	-	+	+
Approches Objet	++	+	+	+	+	+
Approches logiques	++	-	-	-	++	-
Approches ontologiques	++	++	+	+	++	+

TABLE 2.1 – Niveau d'exploitation des paramètres

C : Compositionnalité
P : Partialité
R : Richesse des informations
G : Gestion de l'incomplétude
N : Niveau de formalisme
A : Applicabilité à l'existant

2.2.4 Notion informelle de contexte

Les systèmes informatiques s'efforcent de trouver des représentations formelles de la réalité pouvant être traitées par un langage de programmation. Par conséquent, lorsque l'on utilise la notion de contexte dans des programmes informatiques, il devient essentiel d'en donner une représentation formelle.

Plusieurs chercheurs dans différents domaines se sont penchés sur la représentation du contexte avec souvent des points de vue distincts, mais complémentaires. Par exemple, avec des systèmes issus de l'informatique ubiquitaire, le contexte d'une application est plutôt à considérer en termes de paramètres physiques [58], alors que dans les domaines s'intéressant à l'interaction personne-machine, le contexte est à la fois lié aux tâches des utilisateurs [54] et à l'historique des interactions avec la machine [56].

Par la suite, dans ce mémoire, nous adoptons le point de vue de l'intelligence artificielle, où la notion de contexte s'occupe de la représentation de l'information, ou, plus précisément, de ce qui caractérise l'information elle-même. Nous montrerons ensuite, que le contexte doit être considéré comme un concept et nous en proposerons une description intensionnelle.

Après une étude approfondie des nombreux travaux s'intéressant à la modélisation du contexte, nous avons retenu un petit nombre de propriétés essentielles à notre souci de caractérisation de l'information sur le contexte.

Propriété 1 Explication de la relation entre une situation et un contexte.

La plupart des personnes qui travaillent sur le raisonnement en représentation des connaissances considèrent une situation comme un ensemble de faits explicites, i.e., des relations entre les objets d'une situation. Pour elles, les contextes ne sont pas des situations puisqu'une situation enregistre l'état du monde tel qu'il est, indépendamment de tout objectif à atteindre par un agent, alors qu'un contexte, lui, doit être analysé comme étant une théorie du monde encodant une vue subjective d'un agent qui a un but précis à atteindre. Suivant l'idée de [65] nous pouvons dire qu'il existe quatre manières de lier un contexte à une situation.

- Une situation est un ensemble de contextes et chaque contexte est considéré comme une perspective associé à une partie d'une même situation.
- Des contextes sont associés à des situations par une correspondance un à un entre contexte et situation. Avec cette approche, c'est l'évolution dans le temps de la situation qui correspond à l'évolution du processus de raisonnement [64].
- Un contexte correspond à plusieurs situations. Avec cette approche, le contexte n'est pas pris en compte dans le processus de raisonnement.
- Plusieurs situations sont décrites par plusieurs contextes. Cette approche résulte d'une combinaison des cas précédents.

Par conséquent, l'une des possibilités doit être sélectionnée pour une compréhension claire de la modélisation du contexte.

Propriété 2 De quoi est composé un contexte ?

Quelques éléments de réponse à cette question peuvent être trouvées dans la théorie des situations [1], comme le traitement de la partialité et la capacité à traiter des informations dynamiques. Dans la théorie des situations, l'unité de base de l'information est appelée infon. Il est dénoté par $\iota = \langle\langle R, x_1, x_2, ...x_n, i\rangle\rangle$, où R est une relation à n arguments, ces n arguments x_i peuvent être des individuels (instances), des lieux, des temps, des situations, des types ou des paramètres notés par des variables pointées (e.g., $\dot{t}$). La polarité i indique qu'un item d'information, par exemple un infon est caractérisé par un état de vérité (1 pour vrai et 0 pour faux) dans une situation donnée. Les paramètres pouvant être des variables (i.e., un types de données) et leurs valeurs sont attribuées par les fonctions. Par exemple, l'infon $\langle\langle On, ACsystem, \dot{t}, 1\rangle\rangle$ signifie qu'un système d'air conditionné est spécifié par un paramètre de temps $\dot{t}$ et que nous rechercherons les situations où ce système fonctionne (On). La validation d'un infon dépend de la situation. Pour capturer la sémantique d'une situation, la théorie des situations définit la relation $\models$ (i.e., valide) entre des situations et des infons. En utilisant une notation de la logique du premier ordre, une situation s validant l'infon ι avec $s \models \iota$ signifie que l'infon ι est vrai dans la situation S. Comme la condition de validité d'un infon est relative à une situation, les infons sont dits dépendants de la situation. Les types représentent ce que Barwise appelle les "uniformités" partagées par les infons. Par exemple, les trois infons suivants $\langle\langle Weather, Glasgow, sunny, 1\rangle\rangle$, $\langle\langle Weather, NewYork, sunny, 1\rangle\rangle$ et $\langle\langle Weather, Geneva, sunny, 1\rangle\rangle$ partagent la même information « Sunny ». Ce qui diffère dans ces infons est la ville. Le type correspondant à l'abstraction de ce qui peut être défini par : $\phi = [\dot{s} \mid \dot{s} \models \langle\langle Weather, \dot{c}, sunny, 1\rangle\rangle]$ est le type de toute situation à propos d'une ville (il est représenté dans le type par le paramètre $\dot{c}$ quand le soleil est présent. Si s est l'une de ces situations, ceci s'écrit $s \models \phi$. Si ι_1 et ι_2 sont des infons, avec la même relation R et que ι_2 a au moins les mêmes arguments que ι_1, alors ι_1 subsume ι_2. Ainsi, les infons peuvent représenter des informations partielles et un infon vrai est appelé un fait. Les faits reliés à un contexte donné sont représentés

par des paramètres (les infons) alors que les règles d'un contexte sont représentées par des contraintes. Donc, un contexte est défini comme une combinaison de situations et de règles [12]. Les contextes supportent deux types d'infons, les infons factuels et les contraintes pour les paramètres conditionnels, par exemple :

$$S_1 = [\dot{s} \mid \dot{s} \models \langle\langle HigherTan, \dot{x}, 21, 1\rangle\rangle]$$
$$S_2 = [\dot{s} \mid \dot{s} \models \langle\langle start, \dot{y}, 1\rangle\rangle]$$
$$B \models \langle\langle ACSystem, \dot{y}, 1\rangle\rangle$$
$$C = [S_1 \Rightarrow S_2 \mid B]$$

B est un ensemble de conditions pour lequel la contrainte C exprime une information. S'il existe une situation réelle s, de type S, montrant que la température dépasse 21° Celsius (e.g., S_1), et que la condition B est vraie (i.e., il existe un système d'air conditionné), il est alors possible d'inférer à travers C que le système d'air conditionné peut être démarré (e.g., S_2).
Un résultat significatif de la théorie des situations suggère qu'un contexte peut être composé à la fois par des faits impliquant des objets d'une situation courante et par des contraintes impliquant ce qui est appelé des « régularités », c'est-à-dire des types.
Pour illustrer la manière dont peut être représentée une situation, voici l'étude d'un exemple concret. Soit un individu invité à un séminaire localisé dans le salon particulier d'un hôtel. Comment est-il alors possible de déduire que l'individu est également localisé dans l'hôtel ?

$$S_0 = [\dot{s}|\dot{s} \models << Located, seminar28, Hotel\_Pierre\_le\_Grand, \dot{t}, 1 >>]$$
$$S_1 = [\dot{s}|\dot{s} \models << Located, Pierre, Hotel\_Pierre\_le\_Grand, \dot{t}, 1 >>]$$
$$B = [\dot{s}|\dot{s} \models << Participates, Pierre, seminar28, \dot{t}, 1 >>]$$

La contrainte est donc définie par :

$$C = [S_0 \Rightarrow S_1|B]$$

Si un séminaire se déroule dans un hôtel donné, alors une personne est également dans cette hôtel parce qu'elle participe au séminaire. En modifiant la situation de la manière suivante :

$$S'_0 = [\dot{s} \mid \quad \dot{s} \models << Located, seminar28, Room\_A, \dot{t}, 1 >> \wedge$$
$$\dot{s} \models << Part\_of, Room\_A, Hotel\_Pierre\_le\_Grand, 1 >>]$$

$$S'_1 = [\dot{s} \mid \quad \dot{s} \models << Located, Pierre, Room\_A, \dot{t}, 1 >> \wedge$$
$$\dot{s} \models << Located, Pierre, Hotel\_Pierre\_le\_Grand, 1 >>]$$

Alors, avec les mêmes contraintes, le résultat obtenu est :

$$C = [S'_0 \Rightarrow S'_1|B]$$

L'introduction de la notion de typage (regularities) et l'autoréférence font la force de la sémantique des situations. Au contraire, sa faiblesse réside principalement dans l'absence d'une logique expressive sous-jacente.

Propriété 3 Comment représenter une relation de subsomption entre les contextes.

Les travaux de McCarthy [100] ont mis en lumière le fait qu'un contexte est un objet de première classe (pouvant être l'argument d'un prédicat). La relation de base se note $ist(c, p)$ affirmant que la proposition p est vraie dans le contexte c.

Comme les contextes sont des objets formels, l'axiomatisation des contextes peut être étendue et une relation générale appelée $specializes(c_1, c_2)$ définit la spécialisation entre les contextes qui sont vrais si c_1 à plus d'hypothèses de c_2.

La transcendance entre les contextes implique qu'un contexte c peut être utilisé dans un nouveau contexte c' par l'hypothèse $c' : ist(c, p)$.

En traitement automatique des langues naturelles (TAL) de nombreuses approches établissent une représentation de la situation des discours à l'aide de ce qui est appelé un DRS ("Discourse Representation Structures") [82].

Les théories sur la représentation du discours reposent sur une définition du langage par des modèles théoriques et une sémantique formelle. Les DRS sont graphiquement composées de deux parties : une partie supérieure qui décrit l'univers du discours (i.e., l'ensemble des variables) et une partie inférieure constituée d'un ensemble de conditions. Une variable introduite dans un univers est interprétée comme étant un objet existant. La théorie octroie aux DRS la possibilité d'imbriquer un DRS dans un autre. Cette possibilité d'emboîtement des DRS permet de représenter l'extensibilité. Cette possibilité existe également avec les enregistrements à types dépendants [35] dont le but est le même que les DRS. Par conséquent, étant donné une structure de la connaissance abstraite d'une situation, la possibilité de disposer de structures emboîtées semble un élément important de la représentation des connaissances.

Propriété 4 Expression de la dépendance entre un contexte et une action.

Des auteurs expriment le fait que la notion de contexte n'est pas absolue, mais plutôt relative à quelque chose comme une activité [56, 83, 39], une tâche utilisateur dans le contexte du diagnostic [22], ou une situation physique [58]. Du point de vue ontologique, il est démontré qu'un contexte peut être catégorisé comme un moment [6] universel [55]. Nous nous concentrerons ici sur la relation entre l'action et le contexte qui couvre un grand nombre d'applications. Par exemple, dans l'activité humaine, le contexte d'une activité est évalué en accord avec cette activité. Si cette activité change, le contexte change alors que la situation peut rester la même. Pour cela, il existe une forte dépendance entre l'action et le contexte. Ajoutez à cela, qu'un contexte est continuellement redéfini par les actions successives. Comme il ne peut pas être considéré en dehors de l'utilisation d'une action, la description d'un contexte doit refléter cette dépendance en établissant un lien explicite entre l'action et le contexte.

2.3 Synthèse

Alors qu'il n'y a pas de consensus sur ce qu'est exactement un contexte, la description d'un contexte apparaît néanmoins comme fortement liée à la spécification du concept de situation [132, 56, 29, 115, 88, 106]. De la propriété 1 et suivant le point de vue de beaucoup d'auteurs [58, 95] nous supposerons que plusieurs contextes peuvent coexister pour une même situation. La propriété 2 nous incite à représenter la spécification d'une structure des contextes avec des types. Il est clair, d'après la propriété 3 que la structure de ce contexte doit être emboîtée, ce qui

6. Un moment est un individuel qui dépend d'un point de vue existentiel dépend d'autres individuels.

implique que partant d'un système logique typé le cadre de travail logique choisi doit supporter le sous-typage. La propriété 4, elle, suggère qu'il est nécessaire d'utiliser une structure de contexte prenant en compte le fait que chaque contexte est fortement lié à la définition d'une action. Pour cela, la description du contexte doit pouvoir inclure ce lien avec les actions (nous parlerons alors d'une action contextualisée). Une description du contexte dans le cadre de ce mémoire qui raffine la définition empirique de l'introduction se définit par :

Un contexte est un agrégat de connaissances utilisé par un agent pour exécuter une action contextuelle dans une situation donnée, à un temps t avec un but précis. Une action contextuelle est une fonction qui met en correspondance un ou plusieurs arguments extraits du contexte avec un symbole utilisé par une action.

Notons que l'action dont nous parlons doit être vue comme une action de haut niveau et elle est reliée à une action de bas niveau, qui est implantée par un langage procédural quelconque. L'ensemble des actions est restreint à l'ensemble des symboles de prédicats correspondants à des verbes d'actions (e.g., bouger, prendre, lire, tourner, etc).

3

La théorie des types

Depuis une trentaine d'années le λ-calcul typé suscite de l'intérêt pour ses rapports étroits avec les langages de programmation et le lien qu'il établit entre les notions de programme et de preuve en logique intuitionniste : « la correspondance de Curry-Howard ». Depuis le premier système de types qui était dû à Curry, beaucoup d'autres systèmes développés : citons par exemple le système Automath de de Bruijn, le système F de Girard, la théorie des types intuitionnistes de Martin-Löf, la théorie des constructions de Coquand et Huet, le calcul des constructions étendues de Luo, etc.

Les langages de programmation sont des outils qui permettent d'écrire des programmes en langage machine tout en gardant, autant que possible, le contrôle sur ce que fera le programme tout au long de son exécution. L'écriture des premiers programmes était accompagnée par des commentaires qui indiquaient ce qu'étaient censées faire les instructions : le « programme source ». A la compilation, lors de la génération du code objet, les commentaires étaient oubliés. Ces programmes sont dits de bas niveau, car les commentaires ne sont pas traités par la machine. Dans un langage de plus haut niveau, une partie de ces commentaires est vérifiée par le compilateur, le reste concerne le programmeur. Cette partie vérifiée des commentaires se nomme : typage (on appelle typage, l'association d'une valeur avec son type). Un langage est considéré comme étant de haut niveau si le système de types est riche (les types de données sont des formules spécifiant les propriétés des programmes et ces types peuvent être des formules arbitrairement complexes). A la compilation, les types sont vérifiés puis oubliés avec le reste des commentaires.

Le lambda-calcul typé est utilisé pour servir de modèle mathématique dans cette situation et le rôle du langage machine est joué par le lambda-calcul pur. Les systèmes de types peuvent être tellement riches qu'ils sont utilisés pour la spécification complète de programmes. Par exemple le démonstrateur de théorème développé par l'INRIA : Coq, qui s'appuie sur le calcul des constructions de Coquand, permet de spécifier des programmes et de générer le code CAML zéro défaut correspondant.

Notre intérêt pour le lambda-calcul typé répond à un besoin un peu différent. L'idée est d'utiliser la richesse de représentation fournie par les types (dont les éléments à typer sont fournis par des ontologies de domaine) pour représenter des « types de contexte » associés à des actions, puis, à l'exécution, de déterminer les actions à entreprendre à partir de l'identification des types de contextes vérifiés dans une situation donnée. Mais avant cela présentons quelques éléments liminaires du lambda-calcul et de la théorie des types.

3.1 La logique intuitionniste

The question where mathematical exactness does exist, is answered differently by the two sides ; the intuitionist says : in the human intellect, the formalist says : on paper.

L. E. J. Brouwer [Bro13], p. 56

Des différentes explications que les dictionnaires donnent de la logique, nous pouvons en retenir la définition suivante : science formelle, qui met en évidence soit l'expression de la vérité, soit le cheminement de la déduction ou du raisonnement. Les auteurs d'un ouvrage récent d'initiation à la logique la décrivent comme « la science des lois du raisonnement, des règles de la pensée », qui « s'occupe de la forme des expressions » et « traite de la validité des raisonnements ».

C'est au 19*e* siècle que la logique moderne se constitue en ce sens et que naît la logique symbolique, logique ancrée dans un rapport étroit avec la mathématique.

Il se produisit au 19*e* siècle une discorde sur le fondement des mathématiques entre les acteurs de la communauté scientifique de l'époque. Celle-ci donna naissance à trois courants de pensée :

- un premier courant se réclamait du logicisme avec Frege, Cantor, Dedekind etc.,
- un second courant se prévalait du formalisme avec Hilbert,
- un troisième courant se réclamait du constructivisme avec Kronecker, Poincaré et Brouwer.

Le constructivisme nous intéresse plus particulièrement car il est à la base de la théorie des types dont nous verrons par la suite qu'elle est particulièrement adaptée au raisonnement et à la représentation des connaissances.

3.1.1 Le constructivisme

Le constructivisme est une philosophie "révisionniste" des mathématiques dont l'objectif est de limiter celles-ci aux procédés constructifs. Pour cela, il considère que les objets mathématiques sont des constructions mentales. Les démonstrations mettant en évidence l'existence d'un objet sont appelées des *preuves constructives*. Le mot constructif est bien souvent employé comme un mot équivalent à algorithme (et, par la Thèse de Church : fonction récursive générale).

Pour les constructivistes les mathématiques sont une élaboration de l'homme. Elles ne sont pas déjà là (comme le suggère le point de vue existentialiste). Elles n'adoptent pas comme critère de vérité les hypothèses particulières de chaque mathématicien liées à une métaphysique destinée à disparaître et qui n'exprimerait que l'angoisse de quelques mathématiciens. Les constructivistes montrent de la sorte que la conception constructive ne mutile pas la mathématique classique, mais au contraire l'enrichit.

Cette philosophie repose sur de nombreuses variétés de constructivismes, dont les plus connues sont :

1. Le **prédicativisme** d'Henri Poincaré.
2. L'**intuitionnisme** de Brouwer et Heyting.
3. Le **finitisme** de Hilbert.
4. Le **constructivisme** de Bishop.

5. Le **constructivisme Russe** de Markov.

Brouwer est considéré comme le chef de file du mouvement intuitionniste. Il était soucieux de définir ou de démontrer, qu'il existe un algorithme permettant d'exhiber l'objet mathématique étudié. Mais sa preuve abstraite d'existence ne suffisait pas. Il refusa l'axiome du choix et remis en cause le principe du tiers exclu [7].

3.1.2 L'intuitionisme

"Digging Deeper for the Mole : Intuitionism", soit : creuser, faire des fouilles pour trouver la taupe, qui fait référence à un passage de Hamlet et (n.d.t) au fait que les longues galeries creusées par la taupe sont décelables à la surface par des monticules de terre.

Le mot *Intuitionisme* est un néologisme d'origine britannique entré dans notre langue vers 1850. Il désigne une doctrine privilégiant un mode de connaissance direct et immédiat censé atteindre une réalité individuée et actuellement donnée.

Pour les mathématiques, intuition et construction sont des mots presque équivalents. Les mathématiciens logiciens intuitionnistes désignent plus strictement par "constructifs" des objets réglés, suites ou fonctions, mais aussi, des procédés appelés prédicatifs : des objets définissables à partir d'objets antérieurement définis.

Les travaux de Heyting et ceux de Kolmogorov fournissent à la logique intuitionniste des bases aussi rigoureuses que celles de la logique classique. Elles sont d'ailleurs équiconsistantes : si l'une d'elles est exempte de contradictions alors il en est de même pour l'autre.

Heyting prend le mot *construction* comme terme primitif et emprunte le vocabulaire de Brouwer pour expliciter le sens des connexions logiques en intuitionnisme (1966, rééd. 1971, chap. 7, 7.1.1) :

> – La disjonction p $\vee$ q peut être affirmée exactement si l'une au moins des propositions p et q peut être affirmée... Une proposition mathématique p demande toujours une construction mathématique douée de certaines propriétés ; cette proposition pourra être affirmée pour autant qu'une telle construction ait été effectuée. Nous disons alors que la construction prouve la proposition p et nous l'appelons une démonstration de p. Par raison de brièveté nous désignons encore par p toute construction mise en évidence par la proposition p. Alors

7. La loi logique mise en doute par Brouwer est souvent appelée principe du milieu exclu (principium medii exclusi), ce qui donne faussement à entendre la mise à l'écart d'un tiers cas intermédiaire entre p et non-p. Ce qui est exclu, c'est plutôt l'indétermination soit dans les propriétés soit dans les objets. Kant décrit ce principe comme principe de déterminabilité (pour les concepts), et il le double d'un principe de détermination complète (pour les objets) : "Tout concept, par rapport à ce qui n'est pas contenu en lui, est indéterminé et soumis à ce principe de déterminabilité, à savoir que deux prédicats contradictoirement opposés, un seul peut lui convenir, principe qui lui-même repose sur le principe de contradiction, et qui est donc purement logique, faisant abstraction de tout contenu de la connaissance". D'autre part toute chose est soumise au "principe de détermination complète, d'après lequel, de tous les prédicats possibles des choses, en tant qu'ils sont comparés à leurs contraires, il y en a un qui doit lui convenir". Ce principe, à la différence du précédent, concerne le contenu et non pas seulement la forme logique ; la déterminabilité qu'il affirme ne se restreint pas à des paires de prédicats mutuellement contradictoires, elle se rapporte à l'ensemble de tous les prédicats possibles, ce qui suppose la représentation du tout de la réalité : "pour connaître intégralement une chose, il faut tout le possible et la déterminer par là" (Critique, A 571-573).

¬ p peut être affirmée exactement si nous possédons une construction qui de l'hypothèse qu'une construction p a été effectuée conduit à une contradiction.
- L'implication p → q peut être affirmée exactement si nous possédons une construction r qui, jointe à toute construction qui démontre p (à supposer que cette dernière ait été effectuée), effectue automatiquement une construction qui démontre q. Autrement dit une démonstration de p, prise avec r, formerait une démonstration de q.

N.B. : Le système déductif proposé par Heyting pour la logique intuitionniste est parfois appelé la théorie des constructions.

3.1.3 La sémantique des preuves

There is an old joke which says how to distinguish computer scientists from logicians. You ask : "Do you know what time is it ?". All of them look their watches, but computer scientists say : "4 :12.35, 4 :12.36, 4 :12.37, : : : ", while logicians answer : "Yes". Actually, both answers are right ; but the first one has a constructive aspect : it exhibits some witness. This joke reflects, in a humorous way, the reasoning mechanism of two logical frameworks : Intuitionistic Logic and Classical Logic.

Dans la section suivante nous parlerons d'un principe important qui est à la base de la théorie des types à savoir le fait que les preuves sont des programmes. Ce principe est connu sous le nom de "sémantique de Heyting" ou encore "interprétation de Brouwer-Heyting-Kolmogorov". Il repose sur une perception de la vérité différente de celle communément admise. Pour bien comprendre ce principe de vérité, nous parlerons d'abord du point de vue classique, avec "la sémantique selon Tarski", puis nous présenterons le point de vue intuitionniste avec "la sémantique de Heyting".

La sémantique selon Tarski

Pour Tarski (1902-1985), la vérité est une propriété des énoncés d'un langage. Quelles conditions doivent être satisfaites, lors de la définition d'une propriété d'un énoncé, pour pouvoir dire que ce qui a été défini est la vérité (et non une autre propriété) ?

Tarski stipule la réponse suivante : une définition de "vrai (pour le langage L)" est matériellement adéquate si et seulement si il est possible d'en déduire tous les énoncés de forme

$$E_1 \text{ est vrai (en L) ssi } t$$

où "E_1" est le nom d'un énoncé donné de L, et t sa traduction - supposée préétablie - dans le métalangage dans lequel la définition est formulée (et par rapport auquel L est le langage-objet).

Selon Tarski, nous sommes tous d'accord sur le fait que, si (par ex.) l'énoncé "Platon était un élève de Socrate" est vrai, alors Platon était un élève de Socrate, et réciproquement si Platon était un élève de Socrate, alors l'énoncé "Platon était un élève de Socrate" est vrai.

La théorie de la vérité de Tarski définit d'une manière récurrente une relation dite de satisfaction (à savoir le fait pour une formule ou une proposition d'être vraie) entre d'une part la

formule à évaluer et d'autre part la sémantique selon laquelle la dite formule est évaluée.

Pour Tarski, la dénotation est somme toute plate. Par exemple, le symbole $\wedge$ se traduit par "et". Seule compte la dénotation vrai ou faux. Ainsi : est-grand(Paul) est dénoté par vrai, alors que est-petit(Paul) est dénoté par faux. Pour les expressions, la dénotation est donnée par une table de vérité. Le caractère tautologique de la définition tarskienne de la vérité : Paul est petit ssi « Paul est petit » est le symptôme de l'oubli de la perception dans l'analytique logique classique.

La sémantique selon Heyting

En 1930, Heyting énonça des règles qui de nos jours sont connues sous le nom de "Sémantique de Heyting". Ces règles donnent une définition des énoncés en terme de démonstrabilité. Elles furent le véritable fondement de la logique intuitionniste.

Pour Heyting, l'ambition est de modéliser non pas les dénotations, mais les preuves. Plutôt que de se poser la question "quand un terme [8] est-il faux ou vrai ?", il se pose la question "qu'est-ce qu'une preuve de T ?". Par preuve, il faut comprendre, non pas une transcription syntaxique formelle, mais l'objet inhérent. Par exemple à la figure 3.1 Heyting donne une preuve du prédicat Président(Obama) : Une photo de son investiture.

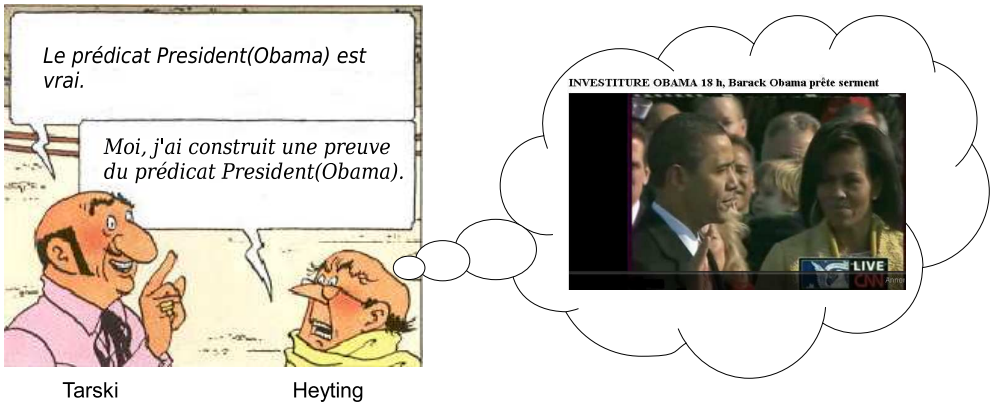


FIGURE 3.1 – Tarski vs Heyting

Donc dans le cadre intuitioniste, la vérité d'une proposition est définie par les concepts de preuve et d'existence. Un jugement de la forme "S *vrai*", où S : *prop*, un jugement qui signifie qu'il existe une preuve de la proposition S :

S vrai = il existe une preuve de S

Le jugement "S vrai" est une manière incomplète et abrégée de dire que s est une preuve de S (ce qui se note $s : S$). Nous parlons d'abréviation parce que la preuve s de S n'est pas donnée effectivement. Seule est donnée l'information qu'il existe une preuve de la proposition en question. Pour déterminer si une proposition S est vraie, il est nécessaire de déterminer les conditions de

8. "terme" signifie "expression représentant une donnée"

preuve ou de réfutation de cette dernière. Le tableau ci-après montre de quelle manière les conditions de vérité des opérateurs logiques changent lorsqu'elles sont conçues comme des conditions de preuve (i.e., sous l'interprétation S vrai = s est un objet de preuve de S, ou s :S). La colonne centrale du tableau présente les conditions de vérité classiques des opérateurs logiques, alors que la colonne de droite présente les conditions de preuve relatives à l'interprétation intuitionniste des opérateurs.

Opérateurs logiques	Conditions de vérité (Tarski)	Conditions de preuve (Heyting)
$\bot$	aucune	aucune
$\wedge$	$\frac{S\ vrai \qquad T\ vrai}{S \wedge T\ vrai}$	$\frac{s{:}S \qquad t{:}T}{(s,t){:}S \wedge T}$
$\vee$	$\frac{S\ vrai}{S \vee T\ vrai} \quad \frac{T\ vrai}{S \vee T\ vrai}$	$\frac{s{:}S}{g(s){:}S \vee T} \quad \frac{t{:}T}{d(t){:}S \vee T}$
$\supset$	$\frac{S\ vrai \vdash T\ vrai}{S \supset T\ vrai}$	$\frac{s{:}S \vdash t{:}T}{(\lambda x)b(x){:}S \supset T}$
$\forall$	$\frac{s \in S \vdash T(s)\ vrai}{(\forall s \in S)T(s)\ vrai}$	$\frac{s{:}S \vdash t{:}T(s)}{(\lambda s)t(s){:}(\forall s{:}S)T(s)}$
$\exists$	$\frac{s \in S \qquad T(s)\ vrai}{(\exists s \in S)T(s)\ vrai}$	$\frac{s{:}S \vdash t{:}T(s)}{(s,t){:}(\forall s{:}S)T(s)}$

Ces preuves sont construites à l'aide de Lambda expressions que nous introduirons à la section suivante.

N.B. : Pierre Cartier [28], pendant le séminaire Bourbaki en 1978 sur la théorie des catégories, apporte une définition plus formelle de la sémantique de Heyting.

Nous parlerons dans la section suivante de l'isomorphisme de Curry-Howard qui permet d'associer des preuves (entièrement formalisées en logique intuitionniste) et le Lambda Calcul qui est un langage de programmation fonctionnelle de très bas niveau. Le lambda-calcul est un cadre naturel pour représenter des preuves et nous montrerons plus particulièrement qu'il existe un lambda-calcul typé avec lequel il est possible de représenter des objets et des relations d'une manière très précise.

3.2 Le $\lambda - calcul$

Le $\lambda - calcul$ est une théorie mathématique des fonctions, issue d'un système logique inventé par Alonzo Church dans les années 30. Il est la partie mathématique du système en question. Le but de Church était de proposer une base pour le fondement des mathématiques et du raisonnement prédicatif. L'idée de prendre la notion de fonction et d'application comme primitive remontait à Frege. Elle est également à la base de la logique combinatoire de Schönfinkel (1924), qui est une théorie du premier ordre équivalente au $\lambda - calcul$.

Le $\lambda - calcul$ permet de manipuler des fonctions indépendamment de leur valeur pour la sémantique. En cela, il procède d'un point de vue intensionnel. En effet, une fonction est définie comme étant l'abstraction d'un nom dans une expression.

Pour illustrer ce propos, l'exemple d'une abstraction du nom w dans l'expression $w + 1$ construit la fonction $w \rightarrow w + 1$, appartenant à l'ensemble $N \rightarrow N$ (espace de fonction des entiers naturels dans les entiers naturels) et qui peut être comprise comme un processus de calcul du successeur. Le $\lambda - calcul$ fut historiquement essentiel pour l'élaboration des notions mathématiques de calculabilité (Church et Kleene). Ainsi, Church a démontré que les fonctions dites λ calculables et les fonctions Turing calculables sont identiques.

Il faut distinguer deux versions du $\lambda - calcul$:

Le $\lambda - calcul\ pur$ qui possède le plus grand pouvoir d'expressivité au niveau des termes.

Le $\lambda - calcul\ typé$ (Curry 1934), qui est régulé au moyen de l'ajout d'une couche logique.

3.2.1 Le $\lambda - calcul$ pur

Le $\lambda - calcul$ pur possède deux opérations de base :

1. L'application notée MN peut être interprétée comme la donnée d'un algorithme M appliqué à un argument N.
2. L'opérateur d'abstraction (λ), suivant $\lambda z.M(z)$ peut correspondre intuitivement à la fonction ensembliste $z \mapsto M(z)$, i.e. que z est assignée à $M(z)$, la variable étant bien sûr libre dans M.

Il faut ajouter à celà une opération de calcul appelée $\beta - conversion$ qui consiste à substituer successivement les arguments aux variables correpondantes.

Définition 4. Les λ-termes.

1. Les λ-termes sont des mots de l'alphabet : $v_0, v_1, ...$ appelés variables, λ appelé l'abstraction, (,) les parenthèses.
2. L'ensemble des λ termes Λ est défini de manière inductive par :
   (a) $z \in \Lambda$;
   (b) $M \in \Lambda \Rightarrow (\lambda x M) \in \Lambda$;
   (c) $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$, où z est une variable quelconque.

La β-réduction et la β-équivalence

Lorsque nous remplaçons les arguments formels de termes par leurs arguments réels et que le résultat obtenu après ce remplacement est identique, nous pouvons affirmer que les deux termes sont équivalents, où plus précisément β-équivalents. Pour être plus formels : il s'agit du en remplacement des sous-termes de la forme $((\lambda\, x\ :\ M\, t)\, v)$ par le terme $t\,[x \leftarrow v]$. La relation entre un terme et un autre obtenu par β réduction (en une étape) sur le premier se note : $t \triangleright u$.

La relation $t \equiv u$ (i.e., t est β-équivalent à u) est définie comme la fermeture réflexive et transitive de la relation $\triangleright$.

Pour ce qui est de ses applications, le $\lambda - calcul$ s'utilise souvent comme support pour la sémantique dénotationnelle [9] ou encore comme langage de spécification. Mais pour cela il devient nécessaire de définir une extension du $\lambda - calcul$ appelé $\lambda - calcul$ typé.

3.2.2 Le $\lambda - calcul$ typé

Le λ-calcul permet de rapprocher les fonctions du λ-calcul des fonctions classiques en associant à chaque abstraction un domaine et un co-domaine.

Un système de types est une *classe de formules* d'un langage qui permet d'exprimer des *propriétés* de termes. Ces formules sont introduites comme commentaires dans les termes qui deviennent alors des "termes typés".

Définition 5 (Propriété de Church-Rosser)**.** Soit R une relation binaire sur un ensemble quelconque ; R a la propriété de Church-Rosser si : $vRw, vRw' \Rightarrow (\exists\, t \,|\, w\, R\, t \,\wedge\, w'\, R\, t)$

Plus clairement, deux termes sont égaux de manière algorithmique, s'ils peuvent être réduits en un terme commun, ce qui garantit l'unicité de la valeur d'un terme, s'il existe. Il est à noter que la $\beta - conversion$ possède la propriété de Church-Rosser.

Définition 6 (Variables typées)**.** Une déclaration de variables est un couple (x,W) où x est une variable du $\lambda - calcul$, et W un type. Cela se note x :W.

Définition 7 (Contexte [10])**.** Un contexte Γ est une application d'un ensemble fini de variables dans l'ensemble des types.

Définition 8. L'ensemble Type de tous les types est défini inductivement de la manière suivante :

1. $0 \in$ Type
2. $\kappa, \iota \in Type \Rightarrow (\kappa \rightarrow \iota) \in Type$

Définition 9. Le $\lambda - calcul$ typé (noté λ^{τ}) est une théorie définie de la manière suivante :

i La donnée d'un alphabet : $v_0^{\sigma}, v_1^{\sigma}, ...$ de variables pour chaque $\sigma \in Type$,
λ,(,) des symboles auxiliaires.

9. La sémantique dénotationnelle est un formalisme algébrique qui permet de spécifier rigoureusement les langages de programmation séquentiels.
10. Nous parlons du contexte logique.

ii Un ensemble de termes de λ^τ du type σ noté Λ_σ est défini inductivement de la manière suivante :
$v_i^\sigma \in \Lambda_\sigma$,
$M \in \Lambda_{\sigma\to\tau}$, $N \in \Lambda_\sigma \Rightarrow (MN) \in \Lambda_\tau$,
$M \in \Lambda_\tau, x \in \Lambda_\sigma \Rightarrow (\lambda x.M) \in \Lambda_{\sigma\to\tau}$ où x est une variable.

iii Les formules de λ^τ sont des équations M = N avec $M, N \in \Lambda_\sigma$ et $\sigma \in Type$

iv λ_τ contient les règles de l'égalité et l'axiome suivant :

$$(\beta)\ \ (\lambda x.M)N = M[x := N]$$

3.3 Les théories des types

Une théorie des types repose sur une ontologie primitive de base et correspond à une catégorisation en classes dénommées types.
Cette ontologie est constituée par des types primitifs d'un domaine de connaissances et par des règles qui déterminent la construction des différents types.

Une théorie des types présuppose que l'on dispose :
– de types primitifs,
– de moyens pour la construction de nouveaux types à partir de types primitifs.

Un type primitif est une catégorisation d'une entité (e.g. dans les langages de programmation les entiers, les booléens, etc.) associée à certaines opérations qu'il est possible de faire sur cette entité ou à partir de cette entité.
Les types primitifs sont composables et donnent naissance à des types dérivés, par exemple, les types cartésiens (i.e. les types de suites finies d'entités de différents types) ou encore les types fonctionnels (i.e. les types d'opérateurs qui construisent des entités d'un certain type à partir d'entités).

Les théories des types sont des systèmes logiques dans lesquels il est possible de représenter des systèmes formels (tel que des théories mathématiques, des langages de programmation ou des systèmes logiques), des propriétés portant sur ces systèmes formels et les preuves de ces propriétés.

3.3.1 Critères d'évaluation

Le concept de machine abstraite fut inventé par Alan Turing en 1936. Grâce à lui, il fut possible de donner une notion précise de la calculabilité qui est à l'origine de " la thèse de Church-Turing ". Le mot thèse est ici utilisé à la place de théorème, puisque la preuve de la notion de fonction calculable par une machine de Turing ne peut pas être démontrée.

Définition 10. Une fonction est dite **calculable** s'il existe un algorithme fini qui permet de calculer, en un nombre fini d'étapes, sa valeur pour tout argument appartenant à son domaine de définition.

Définition 11. Une classe de questions est dite **décidable** s'il existe un algorithme "fini" qui permet de résoudre n'importe laquelle des questions de cette classe en un nombre fini d'étapes.

Définition 12. La propriété de normalisation forte (strong normalisation) indique que tous les termes bien typés sont fortement normalisables (i.e., tout algorithme qui commence avec un terme bien typé se termine).

3.3.2 La correspondance de Curry-Howard

La notion moderne de programme est de nos jours unie à celle de fonction. Depuis fort longtemps l'inconscient des mathématiciens avait fait le rapprochement entre raisonnement et fonction. En effet, le même symbole était déjà employé pour signifier *implication* et *application*. Les premiers systèmes axiomatiques dus à Frege et Russell, étaient plutôt des descriptions du raisonnement mathématique. La justification est venue avec le théorème de complétude de Gödel et Herbrand. Il montre qu'il est possible de décrire les preuves mathématiques d'une manière purement formelle.

Il est aujourd'hui admis que le raisonnement mathématique est complètement mécanisable. Ainsi, il est tout à fait possible de fabriquer des machines à faire des mathématiques avec l'aide de programmes appelés démonstrateurs automatiques de théorèmes. Pour cela une nouvelle sorte de fonction est apparue. Ce sont les fonctions récursives qui sont calculables par une machine (Gödel, Turing, Church, etc.). Pour qu'un ordinateur calcule une fonction, il lui faut un algorithme, c'est-à-dire un programme. Cela nous ramène à la définition des fonctions d'Euler, ainsi une fonction est donnée par une formule mathématique représentant le calcul où le concept de formule a été remplacé par celui de programme.

Mais alors, comment est-il possible d'utiliser les programmes comme preuves ? Les programmes ont une propriété curieuse. En effet, considérés comme des fonctions, leurs arguments et leurs valeurs sont aussi des programmes. Par conséquent, ils forment un monde fermé de fonctions dont il n'est pas possible de sortir. Pour décrire mathématiquement cette situation, il est nécessaire de construire un univers dans lequel les objets, les arguments et les valeurs de fonctions sont représentés par des fonctions. Ainsi, un univers est un type dont les objets sont eux-mêmes des types ou des noms de types. C'est la différence entre un univers à la Russel qui est un type dont les objets sont des types et un univers à la Tarski qui est un type dont les objets sont des noms de types. Dire qu'un univers U est un univers à la Tarski signifie qu'il existe un opérateur, habituellement noté PRF, tel que, chaque fois que $A : U$, alors $PRF(A)$ est un type.

A la fin des années 50, le mathématicien Curry [40] a montré qu'il existait des connexions entre la logique intuitionniste et la logique combinatoire avec des types (sorte $\lambda - calcul$ typé). Les idées développées par Curry reposent en partie sur la sémantique proposée par Heyting pour les connecteurs logiques. Comme nous l'avons expliqué précédemment, pour Heyting une formule ne doit pas être considérée comme vraie, mais elle doit être prouvée, et il en recherche la preuve. Curry montra qu'une manière de trouver la preuve d'une formule était de lui associer un programme (i .e. une fonction). Cette association entre programme et preuve fut appelée isomorphisme de Curry-Howard et a entraîné les conséquences suivantes :

- En logique, elle a permis de développer des programmes pour la démonstration de théorèmes avec de nombreux résultats. L'exemple le plus marquant fut peut-être l'élimination des coupures dans une arithmétique par Girard [63].
- En programmation, elle permet de supprimer un aphorisme courant en informatique "la programmation sans bogues n'existe pas". Pour cela, au lieu d'écrire des programmes, il suffit de donner la démonstration de ces spécifications, puis de transformer la démonstration en programme, qui est alors sans erreurs de construction [78, 11].

La correspondance de Curry-Howard repose sur une tentative d'expliquer la logique à partir

d'un matériau primitif externe au formalisme. Nous n'affirmons pas seulement "une preuve de A ou une preuve de B", nous indiquons aussi laquelle. Cela fait immédiatement, même dans le cas fini, une immense différence avec la sémantique. L'interprétation fonctionnelle ne s'intéresse pas au fait brut de savoir que F est vrai, mais elle explique comment, par exemple dans le cas du *ou*, par la gauche ou par la droite.

Pourquoi lier une logique à des programmes ? La logique permet de garantir une qualité de programme, mais elle permet également de représenter les structures nécessaires à la modélisation de la connaissance. A cet effet, la logique doit être formellement rattachée à un système de types. Dans le cas par exemple d'un langage L, une logique $\mathcal{L}$ peut être définie sur L comme une relation de conséquence $\vdash_{\mathcal{L}}$. Cette logique est caractérisée par un système de règles et d'axiomes Σ^A utilisés pour dériver des jugements de prouvabilité tels que Σ^A prouve $\Gamma \vdash_{\mathcal{L}} \psi$ où Γ représente une séquence arbitraire (éventuellement vide) de propositions et ψ, une proposition arbitraire.

Il existe alors une correspondance du type "propositions comme types" entre, d'une part, un système de preuves Σ^A et sa logique $\mathcal{L}$ et, d'autre part, une théorie des types $\mathcal{T}$ si chaque proposition de $\mathcal{L}$ peut être interprétée comme un type de $\mathcal{T}$. Le type de chaque proposition de $\mathcal{L}$ est dit prouvable dans $\mathcal{T}$. Dans cette correspondance (Curry-Howard-de Bruijn [76]), l'implication joue le rôle d'un type fonctionnel.

Par exemple, dans le cas du modus-ponens, où N et $N \rightarrow M$ sont les prémisses, il faut déduire que M est vérifié. L'isomorphisme de Curry-Howard permet d'associer à N un programme appellé t_N, ainsi qu'un programme $p_{N \rightarrow M}$ à $N \rightarrow M$. De même, le terme qu'il faut construire s'appelle p_M et est calculé par la β-réduction $(p)t$. Le modus ponens correspond donc à l'application d'une fonction à son argument. Le type d'un programme est sa spécification, c'est-à-dire son intention. Il peut donc exister plusieurs programmes pour la réalisation d'un même but.

L'interprétation "formules en tant que types", sur laquelle la théorie constructive des types est fondée, débouche sur d'importantes correspondances : les propositions sont des types et les preuves sont des programmes [6]. Cette dernière suggère une identification des propositions et des types dans le cadre de la logique intuitionniste, de façon à ce que la théorie des types englobe le calcul (intuitionniste) des prédicats, tout en fournissant une alternative à la théorie des ensembles. La théorie des types apporte des objets "preuve" explicites qui sont des termes d'une extension du $\lambda - calcul$ typé. Au niveau syntaxique de la correspondance de Curry-Howard, les formules correspondent à des types, les preuves correspondent à des termes, et la prouvabilité correspond au peuplement, etc.

Etudions maintenant un peu plus en détail ce qu'est la correspondance de Curry-Howard à travers quelques correspondances :

1. La correspondance entre les connecteurs logiques et les constructeurs de types. Cette correspondance est décrite à la figure 3.2. Nous ne présenterons ici que les connecteurs associés à la théorie des types simples puisque nous n'avons pas encore introduit les types dépendants.

   – Le connecteur de conjonction ($\wedge$) correspond au produit cartésien de la théorie des ensembles, il est le type produit ($\times$).
   Par exemple : Prenons $Paul$ et $Jacques$ comme des preuves du type P et $True$, $False$

Théorie de la démonstration	Programmation
Règles logiques (les règles de déduction)	Introduction
Preuves	Programmes
Axiomes, hypothèses	Déclarations de variables
Preuve d'un lemme	Procédures, fonctions
Théorèmes (concl. d'une preuve)	Types, spécification (d'un programme)
Raisonnement par récurrence	Boucles "for"
Réduction d'une preuve (élim. coupures)	Exécution d'un programme
Raisonnement par l'absurde	Instruction d'échappement (e.g. trait. d'erreur)

TABLE 3.1 – Correspondance entre la théorie de la démonstration et les langages de programmation.

comme des preuves du type V. Les éléments du type $P \times B$ sont les paires : (Paul,True), (Paul,False), (Jacques,True), (Jacques,False).

- Le connecteur de disjonction ($\vee$) correspond à l'union disjointe de la théorie des ensembles, il est le type somme ($+$). Ainsi, la somme de $A_1 + A_2 + A_3 + ... + A_j$ contient tous les éléments A_i.
  Par exemple : le type couleur dans un jeu de carte est : *Couleur : Coeur + Carreau + Trèfle + Pique.* Et le type *Carte : Joker + (As* $\times$ *Couleur) + (Roi* $\times$ *Couleur) + (Dame* $\times$ *Couleur) + (Valet* $\times$ *Couleur) + (1,2,3,4,5,6,7,8,9* $\times$ *Couleur).*
- L'implication logique intuitionniste $\Rightarrow$ correspond à l'espace des fonctions de A dans B. Pour les intuitionnistes, l'implication $A \Rightarrow B$ n'est pas égale à $\neg A \vee B$ mais à la construction d'une démonstration de B à partir d'une démonstration de A (contenu calculatoire de la logique intuitionniste). Cette implication correspond au type fonctionnel, qui est défini par le constructeur de type $T_1 \rightarrow T_2$ où T_1 et T_2 sont des types quelconques. Par exemple, le type fonctionnel $Person \rightarrow Age$ représente l'ensemble des fonctions qui prennent en argument une personne (e.g., Jacques) et retournent son âge (e.g., 56 ans).

2. La correspondance entre *supposition* (hypothèse) et *variable* de terme. Lorsque nous écrivons $f : A \rightarrow B,\ x : A$ dans la partie droite du jugement $f : A \rightarrow B, x : A \vdash fx : B$ nous introduisons deux suppositions $f : A \rightarrow B$ et $x : A$. Elles correspondent à des variables dans le $\lambda - calcul$ et à des hypothèses en logique.
3. La correspondance entre *preuves* et *termes*. Tout terme est une construction (i.e., une preuve). Les termes représentent une forme d'arbre de déduction naturelle utilisé pour construire les preuves (termes). Cette analogie explique l'assimilation de l'isomorphisme de Curry-Howard à l'expression "proofs as programs".

Les tableaux 3.1 et 3.2 montrent la correspondance qui existe pour le premier, entre les termes utilisés dans les langages de programmation et les termes employés en théorie de la démonstration et, pour le second entre des symboles de la théorie des types et leurs interprétations en logique (dépendante ou non dépendante), la théorie des ensembles et les langages de programmation.

Les théories typées sont des systèmes logiques avec lesquels il est possible de représenter des systèmes formels (e.g. théories mathématiques, langages de programmation des systèmes logiques), des propriétés de systèmes formels et les preuves de ces propriétés.

Type	Rep. non dép.	Interpr. logique	Interp. log. non dep.	Théorie des ensembles	Langage de programmation
$\varnothing$		$\bot$		ensemble vide	nil(Lisp)
$+$		$\vee$		union disjointe	type concret (ML), type union(C)
Σ	$\times$	$\exists$	$\wedge$	produit cartésien	type enregistrement
Π	$\rightarrow$	$\forall$	$\Rightarrow$	ens. de fonctions	type fonctionnel

TABLE 3.2 – Correspondances entre les types dépendants, la théorie des ensembles et les langages de programmation.

Une distinction doit être faite entre la théorie des types simples et les $\lambda - calculs$ typés plus évolués :

1. Le système F [62], dont l'intérêt réside dans l'utilisation de l'opérateur d'abstraction avec (ordre de), qui permet de disposer d'une syntaxe très simple et d'un grand pouvoir d'expression.
2. La théorie des types de Martin-Löf [98]. Le pouvoir d'expression est plus faible que F, mais avec des possibilités de typage qui permettent d'exprimer plus de nuances que F. Sa syntaxe cependant est complexe.
3. La théorie des constructions de Coquand et Huet, et son extension le Calcul des Constructions Etendu [ECC] [91] forment une synthèse de (1) et de (2). Elle permet d'améliorer le pouvoir expressif de F et introduit les nuances de typage propre au système de Martin-Löf. Toutefois, sa syntaxe reste lourde.

3.3.3 La théorie des types simples de Church ($\lambda^{\rightarrow}$)

La plus élémentaire des théories des types est la théorie des types simples de Church (TTS).

Cette théorie est une formalisation de la logique d'ordre supérieur avec comme langage le λ-calcul simplement typé. La syntaxe de la TTS est composée de deux sortes d'objets :

- Les termes qui dénotent des valeurs et qui sont les termes du $\lambda - calcul$ simplement typé.
- Les types qui représentent des ensembles (non-vides) de valeurs. Ce sont les types du $\lambda - calcul$ simplement typé établis à partir de deux types de base, à savoir le type p des propositions, et le type o des objets élémentaires : la TTS est généralement formée avec les constructeurs de types suivants :
  1. l'espace fonctionnel $\rightarrow$,
  2. le produit fini (1, x),
  3. le coproduit fini (0,x) ou union disjointe.

Soit S une signature multi-sortée avec $T = |S|$ l'ensemble sous-jacent des types, en considérant de plus un ensemble infini $V = v_1, v_2, ...$ de termes appelés variables. Un contexte Γ est un ensemble fini de séquences de variables : $\Gamma = (v_1 : \kappa_1, ..., v_n : \kappa_n)$. En théorie des types, l'expression du fait qu'un terme J est du type κ s'écrit : $\Gamma \vdash J : \kappa$.

Avant de continuer, un petit rappel sur la notion de séquent est nécessaire. Soit L un langage du premier ordre arbitraire, un séquent est une expression formelle $\Gamma \vdash \Delta$, où Γ et Δ sont des

suites finies d'énoncés de L. Le séquent $A_1, ..., A_n \vdash B_1, ..., B_m$, s'interprète par le fait que la conjonction des A_i implique la disjonction des B_j. Plus particulièrement, $\vdash A$ a pour "valeur" A, et $A \vdash$ a pour valeur $\neg A$; quant au séquent vide $\vdash$, il signifie l'absurdité. Il existe une différence entre l'expression des séquents en logique classique où les séquents sont symétriques : $A_1, ..., A_n \vdash B_1, ..., B_m$ et l'expression des séquents en logique intuitionniste où les séquents sont asymétriques : $A_1, ..., A_n \vdash B$. En effet, le système de la logique intuitionniste est obtenu à partir de la logique classique avec une restriction aux séquents qui ne possèdent au plus qu'une formule à droite.

Pour typer un séquent de base, la théorie utilise les deux constructeurs de base suivants :

$$\frac{}{x_1 : \sigma \vdash x_1 : \sigma}\ \text{Identité}$$

$$\frac{\Gamma \vdash L_1 : \sigma_1 \qquad ... \qquad \Gamma \vdash L_n : \sigma_n}{x_1 : \sigma \vdash F(L_1, ..., Ln) : \sigma_{n+1}} \rightarrow \text{(Le symbole de la fonction)}$$

ainsi, que les règles structurelles suivantes :

$$\frac{x_1 : \sigma_1, ..., x_n : \sigma_n \vdash N : \tau}{x_1 : \sigma_1, ..., x_n : \sigma_n, x_{n+1} : \sigma_{n+1} \vdash N : \tau}\ \text{Affaiblisement}$$

$$\frac{\Gamma x_n : \sigma, x_{n+1} : \sigma \vdash N : \tau}{\Gamma x_n : \sigma \vdash N[x_n / v_{n+1}] : \tau}\ \text{Contraction}$$

$$\frac{\Gamma x_i : \sigma, x_{i+1} : \sigma, \Delta \vdash N : \tau}{\Gamma x_i : \sigma, x_{i+1} : \sigma, \Delta \vdash N : [v_i / v_{i+1}, vi + 1 / v_i]}\ \text{Permutation}$$

Il est utile de rappeler que les règles structurelles sont des règles d'inférence qui ne font références à aucun connecteur logique, mais opèrent directement sur les jugements et les séquents.

3.4 La notion de types dépendants

Un prédicat de la théorie naïve des ensembles est une entité "non saturée" ou "incomplète", et tout prédicat peut s'appliquer à toute entité [11]. La conjonction de ces deux principes autorise la formulation d'une proposition contradictoire appelée "paradoxe de Russel" [12]. Pour éviter ce paradoxe, deux voies peuvent être empruntées, ce qui consisterait à abandonner l'un des deux principes au choix. La voie choisie mène alors à deux langages différents, la théorie des ensembles de Zermelo-Fraenkel et la théorie des types.

L'idée de départ amenée par Russel et Whitehead [119] fut d'abandonner le principe selon lequel tout prédicat peut s'appliquer à tout objet - et d'interdire des expressions comme $x \in x$ ou $x \notin x$ -. En décidant de distinguer les objets selon leur degré de fonctionnalité, Russel décrivait des objets de "type" 0, les données naturelles, les ensembles d'objets de "type" 0. Les propriétés de ces objets sont de "type" 1, etc.

Dans la théorie des types dépendants, un terme variable $x : \sigma$ peut être employé pour définir un autre, tel que $\tau(x) : Type$, avec un type dépendant il est possible de décrire l'ensemble des entiers naturels de 1 à n de la manière suivante :

$$n : N \vdash Nat(n) : Type$$

N.B. : Il existe également des types dépendants de types (comme des ensembles dépendants d'ensembles).

Les types dépendants furent pour la première fois étudiés par de Bruijn lors de ses travaux sur le projet AUTOMATH à la fin des années 60. Avec un groupe de chercheurs, il voulait alors créer un langage mathématique pouvant être vérifié par un ordinateur. Plus tard, dans le courant des années 1970, Per Martin Löf (PML) proposa un calcul comparable. Son objectif n'était pas de mécaniser le raisonnement et la preuve, mais de créer un langage pour le fondement des mathématiques constructives. De nos jours, le nom de « théorie des types de Martin-Löf » est utilisé pour parler des théories des types dépendants du premier ordre en général. La théorie de PML fut souvent utilisée comme base pour des outils de preuve automatique comme par exemple NUPRL [34], ALF [96], Veritas [72] ou encore PVS [109].

3.4.1 Les types dépendants du premier ordre

La théorie prédicative des types de PML restreint la quantification par l'utilisation d'une hiérarchie d'univers de types. La quantification sur les types d'un univers forme un nouvel univers d'ordre supérieur. Il n'est pas possible à partir d'univers inférieurs de quantifier sur des univers d'ordre supérieur. Des types de base calculables, comme par exemple les entiers, sont définis dans un univers de base appelée U_0 qui est clos pour les formules logiques. De plus, des formules qui

11. Un prédicat est un syncatégorème particulier, i.e., une expression "incomplète", représentant une fonction d'un domaine D d'objets dans l'ensemble des valeurs vrai, faux dans la logique du premier ordre. Ceci afin de départager les objets auxquels s'applique le prédicat des objets auxquels il ne s'applique pas, engendrant ainsi "l'extension du prédicat" contenue dans le domaine D.

12. L'ensemble des ensembles n'appartenant pas à eux-mêmes appartient-il à lui-même ?
- Oui : comme par définition les membres de cet ensemble n'appartiennent pas à eux-mêmes, il n'appartient pas à lui-même, d'où une contradiction.
- Non : il a la propriété requise pour appartenir à lui-même donc contradiction de nouveau.

quantifient U_0 sont nécessairement du type supérieur U_1. Le reste des univers est construit de la même façon. Bien évidemment la théorie des types indépendants du premier ordre restreint la théorie PML aux seuls univers U_0 et U_1. Dans cette théorie, les types de données et les types logiques peuvent être représentés par une seule théorie et traités d'une manière identique.

Généralement toutes les théories des types sont présentées au moyen des règles de base suivantes (certaines en possèdent d'autres) :

Les règles de formation. Elles expliquent les conditions sous lesquelles il est possible de former (construire) un type. Ce sont des règles syntaxiques.

Les règles d'introduction. Elles expliquent la signification d'un type en indiquant comment il est possible de construire les éléments dits *canoniques* ou normaux de ce type (un type de la théorie constructive des types est toujours construit de manière inductive). Ce sont des règles sémantiques.

Les règles d'élimination. Elles introduisent d'autres éléments du type (dits non-canoniques). Ce sont des règles syntaxiques.

Les types dépendants sont donc des nouveaux types de base :

1. Le Π-type : $\Pi\, x : \sigma \,.\, \tau(x)$, le produit dépendant de $\tau(x)$ ou x est un habitant de σ.
2. Le Σ-type : $\Sigma\, x : \sigma \,.\, \tau(x)$, la somme dépendante de $\tau(x)$ ou x est un habitant de σ.

Un premier exemple de type dépendant est le type des tableaux d'entiers. La taille est le plus souvent représentée dans des langages de programmation par une information indiquée dans le type. Il n'existe alors pas un type *tableau*, mais une famille : $tableau(0)$, $tableau(1)$, ... pour des tableaux de taille 0, 1, ... Prenons une fonction T qui établit une correspondance entre un entier n et un tableau de taille n ne contenant que des 0 :

$$T(0) = []$$
$$T(1) = [0]$$
$$\text{etc.}$$

Cette fonction prend toujours pour argument un entier, par contre le type du résultat de la fonction n'est pas toujours le même. Ce type est $tableau(n)$ où n est la valeur de l'argument de la fonction. Un type possible pour cette fonction est : $entier \rightarrow tableau(n)$. Problème : cette notation n'exprime pas le fait que le n réfère à l'argument de la fonction. Il est nécessaire d'indiquer le nom de l'argument (dans le reste du type) en plus du fait qu'il est du type entier. Ce type peut être représenté par le Π-type suivant :

$$\Pi\, n \,:\, entier \,.\, tableau(n)$$

De cette manière, la notation $E \rightarrow F$ devient un cas particulier de : $\Pi\, x : E \,.\, F$, l'argument x de la fonction n'est pas utilisé dans F et ne demande donc pas à être indiqué.

Ces types dépendants augmentent fortement l'expressivité et la concision de la théorie des types. Ils permettent par exemple de représenter le type complexe $date : Type$ dont un habitant

est une année, un mois et un jour. Certains représentent la date comme un produit cartésien d'entiers naturels $N \times N \times N$ où le premier élément est l'année, le second le mois et le troisième le jour. Il est possible d'améliorer le type en précisant que le nombre des mois n'excède pas 12 et le nombre de jours n'excède pas 31 : $N \times N(12) \times N(31)$.

Seulement voilà, le nombre de jours n'est pas toujours égal à 31 et le mois de février dépend de l'année. Donc une meilleure représentation peut être faite par le type dépendant suivant :

$$date \triangleq \Sigma\, y : N\,.\,\Sigma\, m : Nat(12)\,^{13}\,.\,Nat(\text{nombre de jours pour le mois m de l'année y})$$

où le terme "nombre de jours pour le mois m de l'année y" est définie par un filtre aramétré par les arguments y et m. Un habitant du type date est par exemple : $< 1968, < 3, 10 >>$.

Prenons le cas un peu plus complexe d'une phrase difficile à définir : *Every man who owns a donkey beats it* (« Tout homme qui possède un âne le bat ») se représente par la formule de la logique du premier ordre :

$$\forall d \in Donkey, \forall m \in Man(Owns(m, d) \supset Beats(m, d))$$

Le problème avec cette représentation est que la quantification porte sur tous les hommes et tous les ânes, au lieu de porter uniquement sur les hommes possédant un âne. Cette phrase pourrait également s'écrire :

$$\forall m\, \exists d\, Man(m)\ \wedge\ Donkey(d)\ \wedge\ Owns(m, d)\ \rightarrow Beats(m, d)$$

Mais en écrivant cette formule la quantification existentielle porte sur l'ensemble des ânes, alors qu'elle devait portée sur le sous-ensemble des ânes que possède un homme.

Avec une représentation par des types dépendants il est possible de quantifier uniquement sur un « sous-ensemble » (i.e., chaque âne appartenant à un homme) et l'exemple ci-dessus devient :

$$\Pi d : (\Sigma x : Man.(\Sigma y : Donkey.Owns(x, y)).(Beats(\pi_1 d, \pi_1(\pi_2 d))$$

Dans la théorie des types dépendants, il est également possible d'utiliser les Σ-types pour l'encapsulation :

$$\Sigma f_1 : \sigma_1 \rightarrow \tau_1\,.\,\cdots \Sigma f_m : \sigma_m \rightarrow \tau_m\,.\,Axiome(f_1, \cdots, f_m)$$

où les f_i sont des opérations (programmes) de $\sigma_i \rightarrow \tau_i$ qui satisfont à un axiome (ou spécification) $Axiome(\overrightarrow{f}) : Type$. Les habitants de $Axiome(\overrightarrow{f})$ sont interprétés comme des preuves de cet axiome.

Prenons par exemple, pour un type $\sigma : Type$, la définition de la structure d'un monoïde [14] :

13. L'opérateur Nat construit un type à partir d'un entier, et correspond, par l'isomorphisme de Curry-Howard à un prédicat sur un entier. Ce prédicat est appelé un constructeur de type dont le type est : $\Pi\, x : int\,.\,Nat(x)$.

14. Une opération binaire $*$ sur un ensemble X est dite associative si $(a * b) * c = a * (b * c)$ pour tout triplet a, b, $c \in X$.
Un élément $c \in X$ est dit élément neutre pour l'opération binaire considérée $*$ si $e * x = x * e = x$ pour tout $x \in X$.
Un ensemble X muni d'une opération binaire associative et d'un élément neutre s'appelle par convention un *monoïde*.

$$\text{Monoïde}(\sigma) \triangleq \Sigma op : \sigma \to \sigma \to \sigma \,.\, \Sigma unit : \sigma \,.\, (\Pi x : \sigma \,.\, Eq_\sigma{}^{15}(op\, x\, unit,\, x) \times Eq_\sigma(op\, x\, unit,\, x)) \times (\Pi x, y, z : \sigma \,.\, Eq_\sigma(op\, x (op\, y\, z), op(op\, x\, y) z))$$

Un habitant σ :Monoïde(σ) peut être le tuple $\langle m, \langle e, \langle p, q\rangle\rangle\rangle$ où p est la preuve que $e : \sigma$ est un élément neutre pour l'opération $m : \sigma \to \sigma \to \sigma$, et q une preuve de l'associativité de m.

La théorie des types dépendants d'ordre supérieur qui sera introduite dans la section suivante autorise l'encapsulation des types. Il est alors possible d'écrire un type tel que :
$\Sigma\alpha : Type \,.\, Monoide(\alpha)$. Ces types d'ordres supérieurs peuvent alors être utilisés pour la spécification de modules ou de structures complexes (tels que les enregistrements).

Comment prouver un Π-type ou un Σ-type ?

- Une preuve du type produit $P : \Pi x : \sigma \,.\, \tau$ est une fonction $P \triangleq \lambda x : \sigma \,.\, Px$ qui donne pour chaque élément $M : \sigma$ du domaine de quantification, une preuve $Pm : \tau\,[M/x]$ qui montre que la proposition τ est vraie pour l'élément M.

- Une preuve du type somme forte $P : \Sigma x : \sigma \,.\, \tau$ est une paire $P =< \pi_1 P, \pi_2 P >$ qui consiste en un élément $\pi_1 P : \sigma$ du domaine de quantification et une preuve $\pi_2 P : \tau\,[\pi_1 P/x]$ montrant que la proposition τ est vraie pour l'élément $\pi_1 P$.

Logique intuitionniste	**Théorie constructive des types**
$T_1 \vee T_2$	$T_1 + T_2$
$(\forall x : T_1) T_2(x)$	$(\Pi x : T_1) T_2(x)$
$T_1 \supset T_2$	$(\Pi x : T_1) T_2$ ou $T_1 \to T_2$
$(\exists x : T_1) T_2(x)$	$(\Sigma x : T_1) T_2(x)$
$T_1 \wedge T_2$	$(\Sigma x : T_1) T_2$ ou $T_1\, x\, T_2$

TABLE 3.3 – Correspondance entre la théorie la logique intuitionniste et la théorie constructive des types

3.4.2 Les types dépendants d'ordre supérieur

La théorie des types de PML avec plusieurs univers est une théorie des types dépendants d'ordre supérieur. Il existe également d'autres théories d'ordre supérieur appelées théories des types imprédicatives [63, 117, 37].

Ces théories imprédicatives [16] pour lesquelles deux sortes d'univers sont définies : *Prop* l'univers de la logique et *Type* l'univers des données (ou programmes). Ces théories des types non

15. $Eq_\sigma(x, x')$ est le type de la σ égalité dont x et x' sont des habitants de σ. L'intuition est que ce type est habité ssi x et $x' : \sigma$ sont égaux.

16. Le mot prédicativité [85] fut utilisé et défini pour la première fois par Henri Poincaré, puis par Bertrand Russell [119] pour résoudre le problème des antinomies de la théorie des ensembles et, plus précisément, pour répondre à la question suivante ". Quelles expressions symboliques peut-on regarder comme des définitions ? ". Le problème clef était de résoudre la circularité (des objets pouvaient être constitués par une valeur particulière d'une variable qui intervient dans l'expression de propriétés).
Pour H. Poincaré, lorsqu'une définition n'est pas circulaire, elle est dite prédicative. Dans le cas contraire, elle est dite imprédicative. Russell [120] reprit ce principe pour la définition de ce qu'il appela les fonctions prédicatives, i.e. des fonctions (de niveau n) dont les arguments sont d'un niveau inférieur (niveau n-1) à la fonction elle-même.

prédicatives donnent la possibilité de quantifier sur les types pour former d'autres types. Ainsi, dans le calcul des constructions de Coquand et Huet, les termes peuvent être formés en utilisant le quantificateur universel, et cela même sur le type Prop, le type de toutes les propositions. Par exemple, le terme : $(\forall P : Prop.P) : Prop$ est une proposition valide.

Pour éviter les paradoxes, il est nécessaire que le Type des types de propositions ne soit pas lui-même du type Prop, mais du type Sort. A la différence de la théorie des types de PML, dans le Calcul des Constructions (CC) ou avec ECC, une différence existe entre les types des données et les propositions logiques. En effet, dans le calcul des constructions, la coexistence entre des programmes et des preuves d'un côté et des spécifications et des propositions de l'autre est rendue possible par la définition d'une nouvelle Sorte, la Sorte des propositions : Prop.

Le type de cette Sorte est $Type(i)$ pour tout i, et s'exprime par la formule suivante :

$$E, \Gamma \vdash Prop \leq_{\delta\beta\zeta\iota} Type(i) \text{ pour tout } i$$

Définition 13. (Preuve d'un terme, Proposition) Tout type P dont le type est Prop est appelé proposition. N'importe quel terme dont le type est une proposition est appelé une preuve.

Les Prop peuvent être utilisées pour définir des propositions et des preuves de la même manière que la Sorte Type(i) est utilisée pour les spécifications et les programmes.

D'un point de vue général, tout type est une preuve et donc une proposition intuitionniste. Cependant, cette distinction est faite dans CC, où les Prop ne sont pas utilisées dans un calcul pour pouvoir, par exemple, extraire les programmes. Elles sont utilisées pour donner une preuve d'un programme, garantissant ainsi un code avec zéro défaut.

Dans les théories des types où un point essentiel est la définition de l'égalité entre les types, nous distinguons deux posiblités :

- La première possibilité pour déterminer que deux types A et B sont identiques est d'utiliser la convertibilité (ou égalité intensionnelle) :

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : Type}{\Gamma \vdash M : B}(A \cong B) \tag{3.1}$$

  La théorie des types correspondante est dite intensionnelle (e.g., CC/ECC).
- La seconde possibilité est d'utiliser l'égalité propositionnelle :

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash H : A =_{Type} B}{\Gamma \vdash M : B}(A = B) \tag{3.2}$$

  alors une théorie des types qui utilise cette sorte d'égalité est dite extensionnelle (e.g., Théorie de PML).

Au début des années 90, Luo [92] a proposé une extension du calcul des constructions (CC) pour permettre de résoudre quelques problèmes d'ordre pratique posés par la théorie imprédicative (i.e. pas de $\Sigma - types$). Pour cela, il créa une théorie logique unifiée afin de présenter une

logique d'ordre supérieur par un calcul prédicatif [17] des types (et imprédicatif pour Prop) avec laquelle les types de données permettent la représentation des objets de la logique. Cette logique sera à la base du langage développé par la suite.

En effet, le langage de Luo a pour base une théorie des types décidables avec laquelle il est possible de distinguer entre :
- d'une part les types, pour la représentation des données (i.e., les types de base, les structures de données les $[\Sigma - types]$) et des fonctions les $[\Pi - types]$,
- et d'autre part les propositions pour la représentation des contraintes logiques sur les types.

Il est donc possible en utilisant l'isomorphisme de Curry-Howard d'inférer et de dériver les Types (types et prop) de la théorie en réalisant un algorithme écrit dans un langage fonctionnel.

Nous présentons dans le chapitre suivant une très courte introduction au $\lambda - cube$ pour permettre au lecteur de mieux comprendre les liens qui existent entre les différents $\lambda - calculs$ typés .

3.4.3 Extension de la théorie des types simples et le $\lambda - cube$

Les formalismes étudiés dans ce chapitre reposent sur la notion de $\lambda - calcul$ typé. Dans [6], l'auteur a réalisé un comparaison des différents $\lambda - calculs$ typés qui se retrouve dans un schéma appelé λ cube. Le λ cube forme une généralisation de huit systèmes de types, allant du $\lambda - calcul$ simplement typé de Church au calcul des constructions de Coquand et Huet [77]. Il permet de fournir une syntaxe commune des différents calculs.

Ce système de types "standard" est décrit par la formalisation des "Pure Type Systems" (en abrégé PTS) de Barendregt [8].

Le formalisme de PTS

Le formalisme de PTS fut introduit comme une généralisation du λ-cube de Berardi et Terlouw. Il permet de représenter de nombreux λ-calculs typés à la Church. Dans PTS, il ne faut plus parler de termes, types ou sortes, mais uniquement d'expressions définies par la grammaire :

$$\mathcal{T} ::= \mathcal{V} | \mathcal{C} | (\mathcal{T}\mathcal{T}) | \lambda v : \mathcal{T}.\mathcal{T} | \Pi v : \mathcal{T}.\mathcal{T}$$

où $\mathcal{V}$ décrit un ensemble infini de variables et $\mathcal{C}$ un ensemble de constantes. La seule règle de calcul considérée sur les termes est la β-réduction.

Définition 14. les spécifications d'un PTS consistent en un quintuplet $(\mathcal{S}, \tilde{\mathcal{S}}, \mathcal{A}, \mathcal{R}, \tilde{\mathcal{R}})$ où :

1. $\mathcal{S}$ est un sous-ensemble de $\mathcal{C}$, dont les éléments sont appelés les sortes.

17. Le mot prédicativité fut utilisé et défini pour la première fois par Henri Poincaré. Puis il fut repris par Bertrand Russell pour résoudre le problème des antinomies de la théorie des ensembles. Pour être plus précis pour répondre à la question suivante " quelles expressions symboliques peut-on regarder comme des définitions? " Le problème était lié au fait que des objets étaient constitués par une valeur particulière d'une variable qui intervient dans l'expression de la propriété (auto référence). Pour H. Poincaré lorsqu'une définition n'est pas circulaire elle porte le nom de prédicative, dans le cas contraire elle est dite imprédicative. Russell lui reprit ce principe pour la définition de ce qu'il appela des fonctions prédicatives, i.e., des fonctions (de niveau n) dont les arguments sont d'un niveau inférieur (niveau n-1) à la fonction elle-même.

2. $\tilde{\mathcal{S}}$ est un sous-ensemble de $\mathcal{S}$, dont les éléments sont appelés les $\sim$-sortes.
3. $\mathcal{A}$ est un ensemble d'axiomes de la forme : $c : s$ avec $c \in \mathcal{C}$ et $s \in \mathcal{S}$.
4. $\mathcal{R}$ est un ensemble de règles de la forme : (s_1, s_2, s_3) avec $s_1, s_2, s_3 \in \mathcal{S}$, quand $s_2 = s_3$ les règles s'écriront juste (s_1, s_2).
5. $\tilde{\mathcal{R}}$ est un ensemble de règles, dont les éléments sont appelés des $\sim$-règles, de la forme : (s_1, s_2, s_3) avec $s_1, s_2, s_3 \in \mathcal{S}$, quand $s_2 = s_3$ les règles s'écriront juste (s_1, s_2).

Le λ-cube

Le calcul des constructions est décrit par le λ-cube comme une collection de différentes dépendances formant une structure de treillis qui donne la définition d'une famille de huit systèmes qui a comme la base le λ-calcul simplement typé et comme sommet le calcul des constructions. Les collections de systèmes de types définis ne comprennent que la β-réduction et le constructeur de types Pi.

Les huit systèmes du λ-cube sont des éléments d'une famille particulière de PTS où les règles sont de la forme (s_1, s_2, s_3) ; ce qui peut être abrégé en (s_1, s_2).

Définition 15. (λ-cube) Les huit systèmes du λ-cube sont définis par :

- les sortes : $\mathcal{S}_\rfloor = \{*, \Box\}$,
- les axiomes : $\mathcal{A}_\rfloor = \{* : \Box\}$.

Les huit ensembles de règles sont obtenus de la manière suivante :

Soit $\mathcal{R}_\rfloor = \{(\Box, *), (*, \Box), (\Box, \Box)\}$. Un ensemble de règles $\mathcal{R}$ du λ-cube est égal à $\{(*, *)\} \cup \mathcal{P}(\mathcal{R}_\rfloor)$, pour une certaine partie $\mathcal{P}(\mathcal{R}_\rfloor)$ de l'ensemble $\mathcal{R}_\rfloor$.

Graphiquement, les trois éléments de $\mathcal{R}_\rfloor$ s'interprètent comme trois directions d'un espace à trois dimensions qui est représenté à la figure 3.2.

Les trois directions autorisent chacune des dépendances entre des termes et des types (et vice versa) :

- $(*, *)$, le point de départ, exprime la dépendance entre les termes et les termes.
- $(\Box, *)$, le polymorphisme, permet d'abstraire les types par rapport aux termes.
- $(*, \Box)$, exprime la construction de types dépendants de termes.
- $(\Box, \Box)$, permet de faire des calculs au niveau des types et de construire des connecteurs logiques.

Le λ-cube donne une vue synthétique de nombreux systèmes de types parmi les plus étudiés. Le tableau suivant dû à Barendreght en donne une récapitulation :

3.5 Conclusion

Suivant l'idée de PML, la logique se décompose donc en deux couches : la couche épistémologique et la couche ontologique. Ce sont les assertions et les démonstrations qui sont à la base de la notion logique d'épistémologie. La couche épistémologique est conceptuellement prioritaire par rapport à la notion logique d'ontologie, comme la vérité d'une proposition et les relations

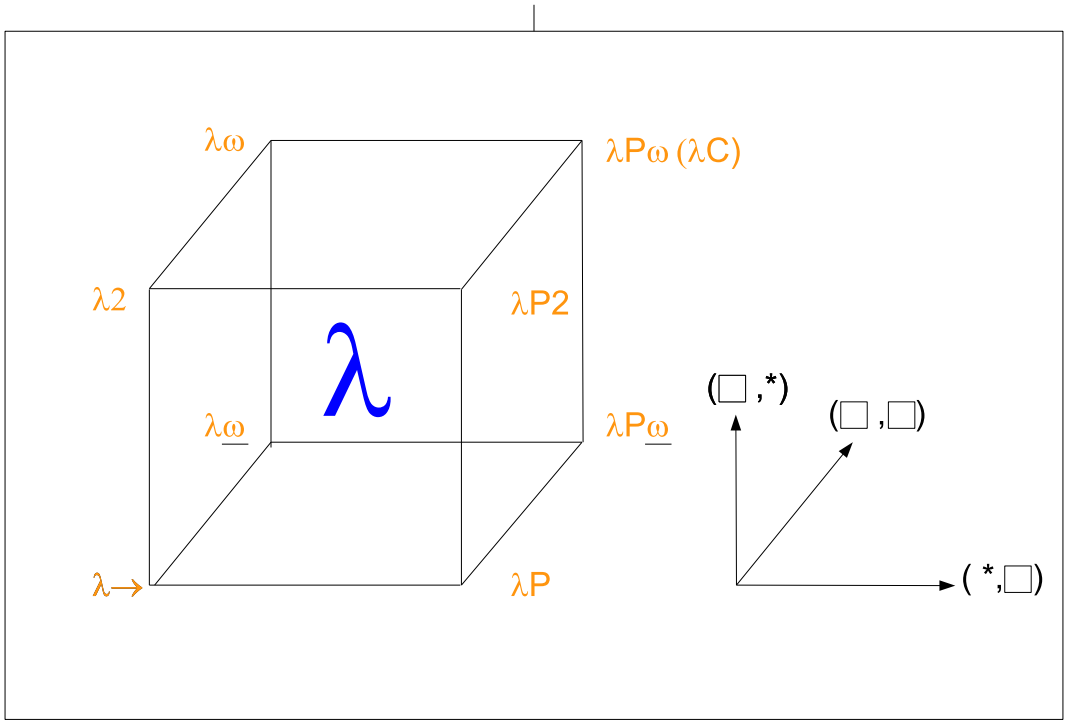


FIGURE 3.2 – Le lambda cube

Systèmes	Règles	Appellation historique
$\lambda \rightarrow$	$(*,*)$	λ-calcul simplement typé
$\lambda 2$	$(*,*),(\Box,*)$	Système F [63]
λP	$(*,*),(*,\Box)$	AUT-QE ; LF [51]
$\lambda P2$	$(*,*),(\Box,*),(*,\Box)$	[89]
$\lambda \underline{\omega}$	$(*,*),(\Box,\Box)$	POLYREC [48]
$\lambda\omega$	$(*,*),(\Box,*),(\Box,\Box)$	F_ω [63]
λC	$(*,*),(\Box,*),(*,\Box),(\Box,\Box)$	F_ω CC [37]

de conséquences entre les propositions. Une logique purement ontologique comme le Tractacus de Wittgenstein est impossible, parce que la notion de base de l'épistémologie est présupposée dans l'explication sémantique des notions ontologiques. Les logiques d'Aristote et de Bolzano avaient le grand mérite, par comparaison avec les logiques modernes, d'inclure à la fois une couche ontologique et une couche épistémologique. Mais la priorité conceptuelle entre les couches est contraire à l'ordre dans lequel l'explication sémantique est donnée (ces logiques subordonnent l'épistémologie à l'ontologie).

En adoptant le point de vue philosophique de PML, le travail développé dans ce mémoire cherche à intégrer la couche ontologique et la couche épistémologique de la logique à celles relatives à la représentation des connaissances. L'ensemble est formalisé par une théorie basée sur une logique d'ordre supérieur.

Nous expliquerons dans la partie suivante comment à partir d'une extension d'une théorie à

types dépendants, il est possible de répondre aux problèmes du raisonnement sur les contextes et les actions. Cette extension, appelée Dependent Type Framework (DTF), étend la théorie de base en définissant la sémantique de deux nouveaux types : le contexte et l'action et en exploitant le sous-typage des contextes tout en conservant la décidabilité de la théorie de base.

4

Conclusion de la partie 1

Pour conclure, les modèles centrés sur les ontologies sont parfaitement adaptés à la représentation des connaissances du domaine, mais beaucoup moins pour raisonner et pour gérer l'aspect dynamique du contexte. Au contraire, le principal point faible des modèles à base logique se situe au niveau de la représentation partielle des connaissances, et également de la dynamicité. A la lumière de ces résultats, il semble qu'une approche conceptuellement optimale du contexte serait celle qui est simultanément capable d'interagir avec une base de connaissances - ontologies - tout en construisant une représentation dynamique du contexte sur une base logique.

Dans la partie 2, il sera montré comment La théorie des types qui associe une forte capacité de raisonnement (par typage) à une grande expressivité obtenue grâce à l'utilisation des types dépendants permet de répondre à cette attente.

il est possible d'utiliser la théorie des types à cette fin.

Deuxième partie

Vers une théorie du contexte et de l'action

1

Présentation du problème

1.1 Introduction

Ce chapitre introduit une formalisation de la notion de contexte différente des représentations classiques, qui prend ses sources dans la théorie intuitionniste des types et la théorie des situations.

Pour commencer, voici un rappel de quelques contraintes logiques et ontologiques que doit satisfaire la représentation des contextes. Dans la partie précédente, il a été précisé que la notion de contexte n'était pas absolue, mais plutôt relative à une activité [56], un diagnostic [22] ou une situation physique [58]. Du point de vue ontologique, un contexte doit être perçu comme un "moment universel"[55], i.e. un concept dont l'existence dépend de concepts intentionnels. Comme un contexte ne peut pas être considéré indépendamment de son intention, il est préférable de l'appeler "contexte-de". Nous avons montré dans la partie précédnte que le lien entre un contexte et une intention s'exprime par une dépendance fonctionnelle représentée par un type dépendant (cf. Chapitre 3, Partie 1).

La théorie des situations a défini un type particulier appelé "abstraction type", qui représente un contexte par un type de situation composé d'infons et de contraintes. La représentation adoptée ici est conforme à cette manière de faire puisqu'elle va associer des faits - ou propositions - avec des contraintes par un mécanisme de typage dépendant inclus dans des briques de connaissance (à la manière des infons).

Dans la première partie, nous avons montré qu'il existait deux perspectives pour représenter un contexte : l'une logique et l'autre ontologique. La modélisation des contextes que nous proposons dans ce mémoire cherche à unifier ces deux perspectives en réalisant une association entre la théorie des types dépendants et les ontologies. Ainsi, l'apparente contradiction qui existe entre l'aspect général de la représentation par des ontologies et la localité des contextes est résolue en utilisant la connaissance du domaine. Le contexte est représenté en utilisant un mécanisme de typage, de sous typage et de dépendance fonctionnelle. Nous allons expliquer comment construire des contextes à partir d'une ontologie de domaine par une théorie des types dépendants : le Calcul des Constructions Etendu (ECC) de [91].

Le fondement de ce calcul repose sur les quatre points suivants :

1. La logique intuitionniste est une base pour le raisonnement constructif sur des objets du

monde.

2. La correspondance de Curry-Howard permet d'associer un mécanisme de typage à une logique d'ordre supérieure.
3. Il existe une distinction formelle entre les types - les concepts - et les objets - les instances.
4. L'utilisation d'une logique d'ordre supérieur - ECC - garantit un haut niveau d'expressivité.

Comme signalé par [138], la théorie des types fournit un modèle formel clair et approprié à la formalisation des ontologies et des contextes [7] dans ECC.

1.2 Le modèle de représentation

La formalisation de la notion de contexte a permis de démontrer qu'il existe une forte interconnexion entre les trois concepts que sont le contexte, l'ontologie et la logique, comme indiqué à la figure 1.1, ainsi dans ce mémoire :

- Les contextes sont représentés par une structure logique appelée "somme dépendante".
- Les contextes sont associés à des briques de connaissance appelées propriétés et/ou contraintes.
- Les structures logiques possèdent une interprétation épistémologique qui permet de les associer à des propriétés ou des contraintes.

Le triangle sémantique projette ces descriptions sur des preuves (intuitionnistes) qui témoignent de leur existence.

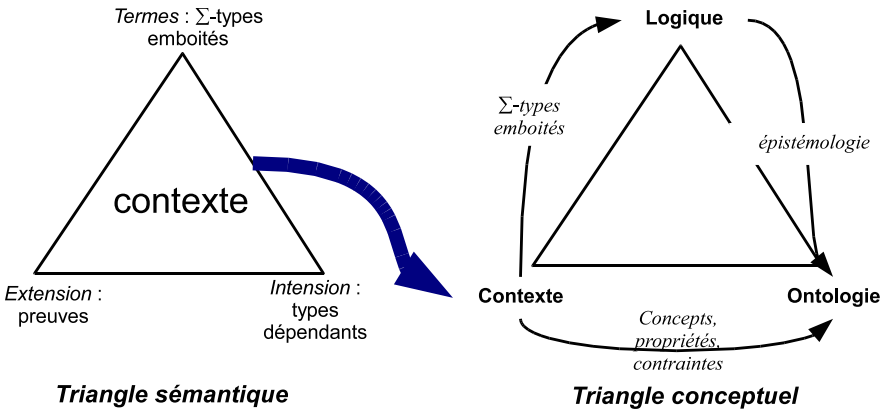


FIGURE 1.1 – Triangle conceptuel et triangle sémantique.

En préambule à l'exposition du modèle développé, une courte explication des choix retenus pour la représentation des contextes et des situations s'impose :

Premièrement, un emprunt est fait à la théorie des situations [52] en adoptant quelques hypothèses faites à propos de la spécification des contextes [1]. La propriété 2 a montré que ce processus d'obtention d'un type est appelé "type abstraction". En suivant ce procédé, un contexte est défini comme un type de situation supportant à la fois des infons factuels et des contraintes.

Deuxièmement, en adoptant la théorie constructive des types, ce travail tente d'unifier un système logique (inclus dans une théorie des types) et un système de représentation des connaissances (ontologies). Afin de synthétiser ces différentes notions (types et ontologies), une théorie nommée DTF pour *Dependant Type Framework* a été créée en réalisant une extension d'ECC.

Généralement, les modèles conceptuels d'ontologies sont subdivisés en trois niveaux dont la couche la plus basse est la logique (cf. 1.1.3). Dans notre modèle, la logique interagit avec les deux autres niveaux. Cette théorie définit un niveau épistémologique qui prend en compte des assertions (jugements), des inférences (démonstrations ou justifications) et des structures de données qui acquièrent à ce niveau un sens spécifique. Elle définit également un niveau ontologique qui permet de donner une signification propre à des notions telles que des propositions, la vérité de ces propositions, les relations de conséquences entre les propositions et des relations complexes. Pour finir, un niveau logique est défini interagissant avec les deux précédants niveaux au sens de Cochiarella [33]. Ce niveau fournit un langage suffisamment riche et expressif pour le raisonnement sur des connaissances et sur les structures formelles valides décrivant des parties de situations.

Il faut tout d'abord montrer comment il est possible de construire les contextes à partir de connaissances ontologiques dans la théorie des types.

1.2.1 La théorie de base

Alors qu'en mathématique les preuves des propositions sont considérées comme des objets, pour d'autres domaines les preuves des propositions peuvent être percues comme le résultat d'un événement, à savoir une situation. En outre, les preuves dans un programme peuvent être le résultat d'une requête sur une base de données ou de connaissances, l'existence d'une fonction effectuant une action (e.g. reliée à un actionneur) ou encore le résultat obtenu par un démonstrateur de théorème étant données une assomption sur des entités, des propriétés ou des contraintes. L'état des informations est formalisé comme des séquences de jugements dont la construction est en accord avec certaines règles. Une telle déclaration de séquence finie est appellée une situation.

Tous les types sont considérés comme bien formés et sont introduits par des axiomes. Ces types spéciaux sont appelés des *sortes*. La théorie des types considère trois niveaux de stratification, le niveau Sorte, le niveau Type et le niveau Objet. Pour pouvoir modéliser le monde réél, ces niveaux sont reliés à des niveaux ontologiques.

2

Vers la spécification du cadriciel (framework) DTF

La théorie des types permet d'assigner à chaque entité un type et d'établir une hiérarchie entre les types. Le type d'une preuve peut être utilisé pour contraindre le type d'objets résultant de la combinaison de plusieurs types. Intuitivement, une conceptualisation serait le résultat de la combinaison d'un ensemble de types utilisés pour représenter la structure d'une situation. C'est pour cela, que les Σ-types sont utilisés pour décrire des faits et des propriétés demeurant identiques quelles que soient les preuves des types qui les composent.

La réutilisation du contexte est rendue possible via la notion de type de contexte. L'idée avancée ici est que la dynamicité peut être assurée par le fait que, dans une situation donnée, n'existent que des objets (instances) appartenant à des types de contexte. L'utilisation de types rend le contexte indépendant des applications envisagées. La spécification d'une "structure type" pour le contexte se réfère à la notion d'épistémologie tandis que la notion d'interprétation du contexte fait plutôt appel à la notion d'ontologie. Cette dernière notion s'est trouvée naturellement au coeur de la modélisation des contextes.

Pour pouvoir ensuite raisonner sur les contextes, ceux-ci étant définis par des structures typées, il faut utiliser une logique possédant des propriétés de calculabilité et capable d'interagir avec un système de typage. Ces propriétés se retrouvent dans une théorie des types via l'isomorphisme de Curry-Howard qui établit une correspondance entre la logique et un système de typage. La logique utilisée est une logique intuitionniste qui s'appuie sur un lambda-calcul typé afin d'augmenter son expressivité.

L'ensemble de ces aspects est donc pris en compte dans une approche fonctionnelle suffisamment expressive et comprenant un système de typage dans lequel les objets ou instances tiennent compte de l'aspect dynamique tandis que les types permettent la réutilisabilité, l'échange et le raisonnement par typage.

Ce chapitre introduit les notions de bases et les définitions de ce qui est appelé "Dependent Type Framework" (DTF), un cadriciel qui étend ECC avec les notions de sous-typage et de constante.

2.1 Le Calcul des Constructions Etendu (ECC)

Les systèmes de types imprédicatifs permettent de formaliser une idée importante des types polymorphes à savoir qu'ils autorisent la quantification et l'abstraction sur tous types de données pour former de nouveaux types. Cette possibilité donne plus de force quant à la logique avec laquelle il est possible de représenter la réalité par ces systèmes. Par exemple, le calcul des constructions est en correspondance directe avec la logique des prédicats intuitionnistes d'ordre supérieur. Par contraste, dans les systèmes prédicatifs, il existe une notion interne de prédicats et de relations qui est représentée par des fonctions propositionnelles (dont les valeurs peuvent être des propositions). Un exemple de ce type de données est le constructeur des $\Sigma - types$.

Les univers sont des types non propositionnels spéciaux dont les objets sont des types. Seuls les types d'un univers inférieur peuvent être utilisés par les types d'un univers i donné. Ce principe est appelé principe de réflexion par PML.

Dans ECC, l'univers imprédicatif Prop des propositions est un objet du $Type_0$ et un objet du $Type_i$ est un objet du type $Type_{i+1}$. Ceci peut être formulé d'une manière intuitive par :

$$Prop \subseteq Type_0 \subseteq Type_1 \subseteq ...$$

Les types sont traités comme des citoyens de première classe. Les objets de $Type_0$ sont des individus, les objets de $Type_1$ les classes d'individus, les objets de $Type_2$ les classes des classes, etc. Les fonctions de deux ou plusieurs arguments, comme les relations ontologiques, sont considérées comme des classes de paires ordonnées et des paires ordonnées de classes de classes.
Comme l'imprédicativité n'existe que pour le type $Prop$, le $Type_i$, la hiérarchie des types s'exprime de la manière suivante $Prop, Type_0, ..., Type_{i-1}$ (i.e., que $Type_{i-1}$ à pour Type : $Type_i$). Les règles de formation décrivant la cumulativité entre les univers sont :

$$\frac{\Gamma \vdash a : Prop}{\Gamma \vdash a : Type_i}$$

$$\frac{\Gamma \vdash a : Type_i}{\Gamma \vdash a : Type_{i+1}}$$

Par souci de simplicité, seuls les types et les propositions sont utilisés ici. Ils dénotent respectivement la sorte des types et la sorte des propositions. La logique utilise les définitions inductives pour la représentation des notions imprédicatives comme la quantification d'ordre supérieur [7].

Les univers $Type_i$ sont prédicatifs pour la formation des $\Sigma - types$ et des $\Pi - types$. En accord avec les règles (Π_2) et (σ), pour tout type A de $Type_i$ et toutes familles de types B[x] d'un $Type_i$, $\Pi x : A.B(x)$ et $\Sigma x : A.B(x)$ sont des objets de type $Type_i$. Le type des $\Pi - types$ et des $\Sigma - types$ est toujours un type supérieur au maximum des types utilisés pour le définir. Par exemple, $Type_0 \rightarrow Prop$ est du type $Type_i$ pour $i \geqslant 1$ mais pas du type Prop ou $Type_0$.

Dans ECC, les propositions représentent des formules logiques. Elles sont utilisées pour la représentation des propriétés des objets du langage. ECC hérite du Calcul des Constructions une logique interne des prédicats intuitionnistes d'ordre supérieur. Les propositions sont formées par une seule opération logique : le quantificateur universel qui est représenté par le constructeur Π. La distinction entre un univers imprédicatif (Prop) et une hiérarchie prédicative ($Type_i$) oblige à spécifier deux règles de formation pour chaque type. Ainsi, le constructeur (Π_1) est donné pour les propositions et le constructeur (Π_2) l'est pour la formation des $\Pi - types$.

2.1.1 La description formelle du langage ECC

Les termes du langage

Les expressions de base du langage sont les termes de la théorie des types donnés par la définition suivante :

Définition 16. (Termes)
Les termes sont définis de manière inductive. Ils sont générés par la grammaire suivante :

$$\begin{array}{ll} \mathcal{T} & ::= \mathcal{V} \\ & \mid Prop \mid Type_i \\ & \mid \Pi V : \mathcal{T}.\mathcal{T} \mid \lambda V : \mathcal{T}.\mathcal{T} \mid (\mathcal{T}\mathcal{T}) \\ & \mid \Sigma V : \mathcal{T}.\mathcal{T} \mid pair_{\mathcal{T}}(\mathcal{T},\mathcal{T}) \mid \pi_1(\mathcal{T}) \mid \pi_2(\mathcal{T}) \end{array}$$

Dans cette définition, $\mathcal{V}$ est un ensemble de variables. $Prop$ et $Type_i$, pour $0 \leq i$, représentent les univers de types qui, par comparaison avec la théorie des ensembles, peuvent être compris comme des ensembles de types. Les $\Pi - types$ forment les espaces de fonctions dépendantes et les $\Sigma - types$ peuvent être éventuellement comparés à des enregistrements à types dépendants. La paire représente le produit cartésien et les π_i les opérateurs de projection sur les Σ-types.

Définition 17. (Variables libres)
L'ensemble des variables libres des termes M, FV(M) est défini par :

$$\begin{array}{lcl} \text{FV(x)} & = & \text{x} \\ \text{FV(Prop)} & = & \emptyset \\ \text{FV}(Type_i) & = & \emptyset \\ \text{FV}(\mathcal{Q}\ \text{x :A.M}) & = & \text{FV(A)} \cup (\text{FV(M)} \setminus \text{x}) \text{ pour } \mathcal{Q} \in \Pi, \lambda, \Sigma \\ \text{FV((M NN))} & = & \text{FV(M)} \cup \text{FV(N)} \\ \text{FV}(pair_T\text{(M,N)}) & = & FV(T) \cup FV(M) \cup FV(N) \\ \text{FV}(\pi_i\text{(M)}) & = & \text{FV(M) pour } i \in [1,2] \end{array}$$

Les sous-termes sont définis d'une façon standard, en particulier, T et M sont des sous-termes de $\lambda x : T.M$. Ceci se vérifie de la même manière pour les Π et les $\Sigma - types$. Les constructeurs de termes $Prop$, $Type_i$, $\Pi - types$ et $\Sigma - types$ doivent être compris comme étant des types.

Définition 18. (Types des sous-termes)
La collection des sous-termes TSt(T) du type d'un terme T est ainsi définie :

$$\begin{array}{lcl} \text{TSt}(Prop) & = & \{Prop\} \\ \text{TSt}(Type_i) & = & \{Type_i\} \\ \text{TSt}(\Pi x : A.B) & = & \{\Pi x : A.B \cup TSt(A) \cup TSt(B)\} \\ \text{TSt}(\Sigma x : A.B) & = & \{\Sigma x : A.B \cup TSt(A) \cup TSt(B)\} \\ \text{TSt(T)} & = & \{T\} \text{ pour tous les constructeurs de types.} \end{array}$$

L'ensemble des types des sous-termes propres de T est défini par $TSt(T)\ T$.

Définition 19. (Réduction et conversion)
La réduction ($\triangleright$) et la conversion sont définies par les schémas de contraction suivants :

1. (β) $(\lambda x : V\,.\,T)U \rhd_\beta [U/x]T$
2. (π) $\pi_i(\langle T_1, T_2\rangle_A) \rhd_\pi T_i$ (i=1,2)

Définition 20. (Relation cumulative)
La relation cumulative $\leq$ est définie par la plus petite relation entre des termes :

1. $\leq$ est un ordre partiel respectant les conversions suivantes :
   (a) $si A \simeq B$, alors $A \leq B$,
   (b) $si A \leq B$ et $B \leq A$, alors $A \simeq B$,
   (c) $si A \leq B$ et $B \leq A$, alors $A \leq B$.
2. $Prop \leq Type_0 \leq Type_1 \leq ...$,
3. $si A_1 \simeq B_1 et A_2 \simeq B_2$, alors $\Pi\, x : A_1\,.\,A_2 \leq \Pi x : B_1.B_2$,
4. $si A_1 \simeq B_1 et A_2 \simeq B_2$, alors $\Sigma\, x : A_1\,.\,A_2 \leq \Sigma\, x : B_1\,.\,B_2$,

De plus, $A < B$ si et seulement si $A \leq B$ et $A \not\simeq B$.

Le typage

Définition 21. (Contexte logique)
Un contexte logique est une séquence finie d'expressions de la forme x :T, où x est une variable et T est un terme. Le contexte vide est défini par $\langle\rangle$. L'ensemble des variables libres d'un contexte $\Gamma \equiv x_1 : A_1, ..., x_n : A_n, FV(\Gamma)$ est défini par $\bigcup_{1\leq i\leq n}(x_i \cup FV(A_i))$

Définition 22. (Jugement)
L'expression d'un jugement est de la forme : $\Gamma \vdash T : V$, où Γ est un contexte, T et A des termes. L'expression se lit "T est du type V dans Γ. Un jugement $\Gamma \vdash T : V$ est appellé *non hypothétique* si Γ est un contexte qui n'est pas vide. Il est dit *hypothétique* dans le cas contraire.

Définition 23. (Dérivation)
La dérivation d'un jugement G est une séquence finie de jugement $G_1, ..., G_n$ avec $G_n \equiv$ G de manière que pour tout $1 \leq i \leq n, G_i$ soit la conclusion de toute instance de la règle d'inférence dont les prémisses sont dans l'ensemble $\{G_j \mid j < i\}$.
Un jugement G est dérivable s'il existe une dérivation de G. Un jugement dérivable s'écrit $\Gamma \vdash T : V$ pour tout terme T.

N.B. : Toutes les règles d'ECC sont des règles de dérivation de jugement. Les différentes règles

d'inférences d'ECC sont :

1. *Les règles de base.*

$$\frac{}{<> \; valid}\ (\text{CEmpty})$$

$$\frac{\Gamma valid}{\Gamma \vdash Prop : Type_i}\ (\text{Prop})$$

$$\frac{\Gamma valid}{\Gamma \vdash Type_i : Type_{i+1}}\ (\text{Type})$$

$$\frac{\Gamma \vdash A : Type_i}{\Gamma, x : A valid}\ (\text{Cval } (x \notin FV(\Gamma)))$$

$$\frac{\Gamma, x : A \qquad \Gamma' valid}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\text{Var})$$

Les règles ($Cempty$) et ($Cval$) formalisent la construction des contextes. Les règles de cumulativité entre les types d'univers sont exprimées par les règles ($Prop$) et ($Type$). La règle (var) décrit le typage des variables.

2. *Les règles de formation pour les propositions.*

$$\frac{\Gamma, x : A \vdash P : Prop}{\Gamma \vdash \Pi x : A . P : Prop}\ (\Pi\text{-form1})$$

$$\frac{\Gamma \vdash A : Type_i \qquad \Gamma, x : A \vdash B : Prop}{\Gamma \vdash \Sigma x : A . B : Type_i}\ (\Sigma\text{-form1})$$

Les règles (Π-Form1) et (Σ-Form1) sont des règles de formation de types, spécifiant comment les éléments des types d'univers $Prop$ peuvent être générés.

3. *Les règles de formation pour les types.*

$$\frac{\Gamma \vdash A : Type_i \qquad \Gamma, x : A \vdash B : Type_i}{\Gamma \vdash \Pi x : A . B : Type_i}\ (\Pi\text{-form2})$$

$$\frac{\Gamma \vdash A : Type_i \qquad \Gamma, x : A \vdash B : Type_i}{\Gamma \vdash \Sigma x : A . B : Type_i}\ (\Sigma\text{-form2})$$

Les règles (Π-Form2) et (Σ-Form2) sont des règles de formation de types, spécifiant comment les éléments des types d'univers $Type_i$ peuvent être générés.

4. *Les règles d'introduction.*

$$\frac{\Gamma, x : A \vdash P : B}{\Gamma \vdash \lambda x : A . P : \Pi x : A . B}\ (\Pi\text{-intro})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B\,[M/x]}{\Gamma \vdash < M, N >} \text{ (Σ-intro)}$$

Les règles (Π-intro) et (Σ-intro) sont des règles d'introduction qui indiquent comment les éléments canoniques des types dépendants respectifs sont formés.

5. *Les règles d'élimination.*

$$\frac{\Gamma \vdash M : \Pi x : A\,.\,B \qquad \Gamma, N : A}{\Gamma \vdash MN : B\,[N/x]} \text{ (Π-elim)}$$

$$\frac{\Gamma \vdash \sigma : \Sigma x : A\,.\,B}{\Gamma \vdash \pi_1\sigma : A} \text{ (π_1-elim)}$$

$$\frac{\Gamma \vdash \sigma : \Sigma x : A\,.\,B}{\Gamma \vdash \pi_2\sigma : B\,[\pi_1\sigma/x]} \text{ (π_2-elim)}$$

Les règles d'élimination (Π-elim), (π_1) et (π_2) définissent comment les éléments des Π-types et des Σ-types peuvent être appliqués.

6. *La règle de sous-typage.*

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A' : Type \qquad A \leq A'}{\Gamma \vdash M : A'} \text{ (Sub)}$$

7. *L'égalité computationnelle.* Le comportement computationnel est achevé par la β- la π-réductions comme cela :

$$(\lambda x : A.M)N \quad \triangleright \quad M[N/x] \tag{2.1}$$
$$\pi_i(\langle M_1, M_2\rangle_A) \quad \triangleright \quad M_i (i = 1, 2) \tag{2.2}$$

Deux termes M et N sont appelés convertible, ce qui s'écrit $M \simeq N$, si ils peuvent être transformés en un autre en répétant l'applications de la β- la π-reductions [90].

$$\frac{\Gamma \vdash M \ : \ A \quad \Gamma \vdash A' \ : \ Type \quad A \simeq A'}{\Gamma \vdash M \ : \ A'} \text{ (conv)} \tag{2.3}$$

Lemme 1. [92] La relation de sous-typage est un ordre partiel sur les termes.

La règle de sous-typage subsume la règle de conversion. La méta propriété suivante de ECC, présentée comme un corollaire nous intéresse également :

Corollaire 2. [92] La relation $\leq$ est le plus petit ordre partiel sur les termes respectant la conversion tel que :
si $A \leq A'$ et $B \leq B'$, alors $\Sigma x : A.B \leq \Sigma x : A'.B'$.

Définition 24. A est appelé le type principal de M dans Γ si et seulement si :

- $\Gamma \vdash M : A$ et
- pour tout $A', \Gamma \vdash M : A'$ ssi $A \leq A'$ est un type de Γ.

Propriétés d'ECC

Définition 25. (Normalisation faible et forte)

- Un terme est faiblement normalisable pour une relation de réduction si pour tout terme t, il existe une séquence finie d'étapes de réduction partant de t et aboutissant à une forme normale de t.
- Un terme est fortement normalisable pour une relation de réduction si pour tous les termes t, toutes les séquences de réduction de t sont finies.

Proposition 3. (Normalisation forte [91])
Le calcul ECC est fortement normalisable.

Définition 26. (Inférence de type, vérification et recherche d'instances d'un type)

i (Inférence de type) Etant donné un contexte Γ et un terme t, l'inférence d'un type est un problème qui consiste à trouver un terme T tel que $\Gamma \vdash t : T$ ou à montrer qu'il n'en n'existe pas.

ii (Vérification de type) Etant donnés un contexte Γ et des termes t et T, la vérification de type est un problème qui consiste à déterminer lesquels des $\Gamma \vdash t : T$ sont vérifiés.

iii (Recherche de l'instance d'un type) Etant donnés un contexte Γ et un type T tels que $\Gamma \vdash T : Type_i$, la recherche de l'instance d'un type est un problème qui consiste à trouver un terme t tel que $\Gamma \vdash t : T$ ou montrer qu'il n'en n'existe pas.

L'inférence de types et la vérification de types sont décidables. En revanche, la recherche d'un habitant pour un type donné (inhabitation problem) est décidable s'il existe un algorithme capable de réaliser cette procédure.

Il est à remarquer que dans ECC nous disposons de deux égalités, une proprositionnnelle (l'égalité de Leibniz) et une computationnelle (la règle conv).

La logique d'ordre supérieur permet de définir une égalité propositionnelle reposant sur le principe de Leibniz, i.e., que deux objets du même type sont égaux s'ils possèdent les mêmes propriétés. Cette idée s'exprime par l'emploi d'un prédicat sur le type des objets comme suit :

Définition 27. (Egalité de Leibniz)
Soit A un Γ-type. L'égalité de Leibniz sur A, notée $=_A$, est une relation binaire sur A définie par :

$$=_A \triangleq \lambda x : A \lambda y : A . \forall F : A \to Prop . F(x) \supset F(y).$$

Il est possible d'écrire $a =_A b$ pour $=_A (a, b)$.

2.1.2 Dualité

La construction de preuves peut s'effectuer soit par dérivation avec les types produit, soit par subsomption avec les types somme. Pour les types produit, prenons le cas simple du produit non dépendant, c'est à dire des fonctions au sens classique ($\to$). La règle d'introduction de la fonction $\to$ est fournie par la $\lambda - abstraction$. Nous pouvons par exemple montrer que les règles

(Π-intro) et (Π-elim) dans le cas de fonctions dépendantes se ramènent aux règles ci-dessous (non-dépendance).

$$\frac{\Gamma, x : A \vdash P : B}{\Gamma \vdash \lambda x : A.P : A \to B}\ (\to -\mathsf{intro}) \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\to -\mathsf{elim})$$

Considérons la dérivation de la tautologie $A \to ((A \to B) \to B)$. Nous obtenons l'arbre de dérivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A,\ y : A \to B \vdash y : A \to B}\ (\text{Var}) \quad \dfrac{}{x : A,\ y : A \to B \vdash x : A}\ (\text{Var})}{x : A, y : A \to B\ \vdash (yx) : B}\ (\to \text{elim})}{x : A\ \vdash \lambda y : A \to B.(yx) : (A \to B) \to B}\ (\to \text{intro})}{\vdash \lambda x : A.\lambda y : A \to B).(yx) : A \to ((A \to B) \to B)}\ (\to \text{intro})$$

Si nous avons des preuves des types de départ ($x : A,\ y : A \to B$), alors nous en déduisons par dérivation que le type racine (en bas) est valide. La dérivation est obtenue en utilisant les règles de la théorie.

Pour le type somme, nous obtenons un arbre de sous-typage (inversé par rapport au précédent) dans lequel chaque noeud est un Σ-type. A la différence du précédent, nous partons du type racine (vide) et nous descendons dans les noeuds. Cette fois les preuves des Σ-types ne sont pas fournies par les règles, mais par l'existence des éléments de chaque paire (dans une base de données par exemple). Lorsqu'un noeud est prouvé, le sous-typage implique que tous les noeuds supérieurs jusqu'à la racine soient prouvés. Nous obtenons ainsi deux arbres duaux avec une construction

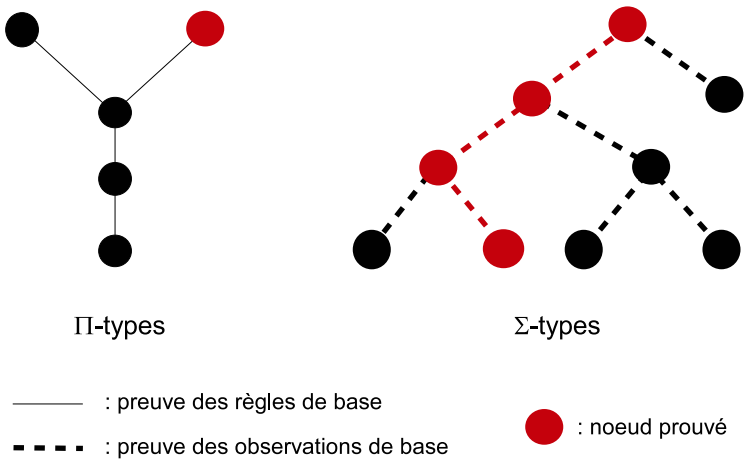


FIGURE 2.1 – Propagation des preuves pour les types dépendants.

duale de preuves (soit par inférence, soit par observation). Supposons que le noeud rouge soit prouvé. Pour le type produit, la preuve se propage si les noeuds supérieurs sont prouvés tandis que pour le type somme, la preuve d'un noeud valide automatiquement tous les noeuds supérieurs. Nous retrouvons là une forme de dualité bien connue des catégoriciens (e.g., produit/co-produit, limite/co-limite, ...).

2.1.3 ECC et le λ-cube

Un sous-système de ECC peut être caractérisé par le PTS suivant :

$\mathcal{S} = \{*, \Box_i\}$

$\mathcal{A} = \{* : \Box_0, \Box_i : \Box_{i+i}\}_{i \in \omega}$

$\mathcal{R} = \{(*,*), (*, \Box_0), (\Box_0, *), (\Box_i, \Box i + i), (\Box_i, *)\}_{i \in \omega}$

où ω désigne l'ensemble des ordinaux.

Le système ECC est plus compliqué puisqu'il permet de créer des empilements de types.

Remarque : ECC ne permet pas d'empiler des variables.

2.2 Notions de base

Comme cela a été expliqué dans la première partie, il existe en logique deux sémantiques pour exprimer la vérité. Le cadriciel DTF est réalisé en prenant pour base la sémantique des preuves dont le pouvoir d'expression permet de représenter la connaissance utilisée par une application d'une manière plus précise (voir par exemple [108]).

Nous ne voulons pas présenter d'une manière philosophique la connaissance, mais simplement identifier quelques caractéristiques de la connaissance à prendre en compte dans le cadre de notre approche. La connaissance est formulée en termes de concept en obéissant à un principe de subjectivité. Il est par exemple possible qu'un agent juge que quelque chose est l'instance d'un certain concept, tandis qu'une autre personne choisira un autre concept. La connaissance est justifiée par le fait que les agents connaissent des choses et ont des raisons pour cela. Comme les connaissances sont structurées et hiérarchisées, les justifications des concepts de base seront associées à des observations ou à des axiomes. Enfin, la connaissance d'un agent peut être étendue (incrémentalité).
Or, si nous partons de l'hypothèse que ces trois caractéristiques (subjectivité, justification et incrémentalité) alors il n'y a pas lieu de faire la distinction usuelle entre connaissance et croyance puisqu'il n'y a pas de connaissance vrai dans l'absolu. Ainsi, chaque concept pourra être décrit par un type dans DTF et chaque instance d'un concept constituera un terme conduisant à une preuve de ce concept. Cela est à rapprocher du travail de [16]. En revanche, l'incrémentalité est associée par l'auteur avec la notion de contexte logique.

En effet, il est tout à fait possible d'améliorer l'expressivité de l'incrémentalité en exploitant judicieusement les Σ-types emboîtés. Pour cela, il suffit d'établir une correspondance avec les ontologies. Les types sont alors employés comme synonymes de concepts. L'individu, lui, représente l'expression de la preuve d'un type simple.

Prédicats

ECC contient une notion interne de prédicat représentée par des types de fonctions propositionnelles. Par exemple : le type des prédicats sur un type A est $A \rightarrow Prop$.

Concepts

Les concepts sont des types qui décrivent des entités au sens général du terme et qui possèdent un critère d'identité ($+I$, les *sortals* au sens de Guarino [71]), ceci afin que la notion d'objet ayant un type possède un sens. Par exemple, le terme $x : Phone$ signifie que la variable libre x est du type $Phone$.

Objets

Un objet est une preuve d'un type, par exemple l'objet $MyNokia6125$ représente mon téléphone portable. Il est une preuve de l'existence d'un objet de type $Phone$.

Propriétés

Les universels, les rôles et les propriétés sont plutôt décrits par des Σ-types. En considérant la somme dépendante $\sigma_1 : \Sigma x : phone.mobile(x))$, une preuve du Σ-type σ_1 est fournie par exemple par la paire $< Nokia6125, q_1 >$ qui précise que pour l'individuel $Nokia6125$, la proposition est prouvée (q_1 est une preuve de $mobile(Nokia6125)$). Si nous nous réfèrons à l'ensemble $\mathcal{B}$ de tous les téléphones, alors les paires prouvées représentent un sous-ensemble de $\mathcal{B}$, le sous-ensemble des téléphones mobiles.

$$< Nokia6125, q_1 >: \Sigma\, x \,:\, phone\,.\,mobile(x)$$

En supplément des Σ-types, deux règles d'élimination (π_1) et (π_2) introduisent les opérateurs de projection π_1 et π_2 tels que $\pi_1 < M, N >= M$ et $\pi_2 < M, N >= N$. Une preuve $s : \Sigma x : T.p$ pour une somme dépendante est une paire $s =< \pi_1 s, \pi_2 s >$ qui consiste en une preuve $\pi_1 s : T$ du domaine du type T et en une preuve $\pi_2 s : p[\pi_1 s/x]$ établissant que la proposition p est vraie pour l'élément $\pi_1 s$. Etant donné que la plupart des relations rencontrées en pratique sont d'arité égale à 2, il est utile de généraliser les types dépendants avec des entrées multiples. Pour un Σ-type agrégé de deux variables d'entrée, la règle d'introduction peut être dérivée de la manière suivante :

$$\frac{\Gamma \vdash L : A \qquad \Gamma \vdash M : B \qquad \Gamma, x : A, y : B \vdash N : P[L/x, M/y]}{\Gamma \vdash < L, < M, N >>: (\Sigma\, x \,:\, A\,.\,(\Sigma\, y \,:\, B\,.\,P))}$$

Par exemple,

$$\Sigma\, x \,:\, Computer\,.\,\Sigma\, y \,:\, WordProcessor\,.\,hasSoftware(x, y)$$

dans lequel $hasSoftware$ est de type $Prop$, représente les ordinateurs dont les logiciels contiennent un traitement de texte.

Inférences

Les inférences sont modélisées par des Π-types. Une instance de $\Pi x : A.B(x)$ est une fonction qui prend comme argument un objet du type A et qui retourne une instance d'un objet du type B. Si B est du type $Type_i$, alors le type de A ne peut pas être dans un univers supérieur à $Type_i$ (prédicativité). Pour les propriétés et les contraintes, B est du type $Prop$ et A peut être défini

dans n'importe quel univers (imprédicativité). Par exemple, le fait que $MyNokia6125$ de type $Phone$ est susceptible d'émettre des appels peut s'écrire avec le produit dépendant suivant :

$$\Pi x : Phone\ .\ sendCalls(x)$$

dans lequel $sendCalls(x)$ est un prédicat dépendant de x selon la règle (Π-intro). La partie calcul apparaît dans la règle d'élimination (Π-elim) où la notation $[N/x]$ précise que toutes les occurrences de x sont remplacées par N. Une preuve du Π-type est fournie par la β-réduction :

$$\lambda x : A.sendCalls(x)(MyNokia6125)$$

c'est à dire : $sendCalls(MyNokia6125)$. En d'autres termes, $sendCalls$ est un prédicat (une fonction $Phone \rightarrow Prop$) qui pour chaque objet x ($MyNokia6125$) du type $Phone$ produit un objet preuve de la proposition ($sendCalls(MyNokia6125)$).

2.3 Les caractéristiques additionnelles

DTF est construit sur ECC par ajout de la notion de constante et de règles de sous-typage. Ces notions étant définies à partir des types existants d'ECC, elles n'affectent en rien la propriété de décidabilité.

2.3.1 L'hypothèse du monde clos

Quelle est la relation entre le langage DTF et l'hypothèse du monde clos ? Dans une base de connaissances ou une ontologie reposant sur une logique booléenne, tous les faits sont soit vrai ($\top$), soit faux ($\bot$). Des faits qui ne possèdent pas de valeur de vérité ne peuvent pas être représentés et "l'hypothèse du monde clos" laisse à supposer que dans ce cas la valeur est fausse (i.e., que l'inconnu est assimilé à la valeur fausse). Du point de vue de la logique intuitionniste, l'hypothèse du monde clos n'a pas de sens. En effet, les faits non prouvés n'impliquent pas le faux (conséquence du rejet par la logique intuitionniste du tiers exclu). Il s'ensuit que les informations positives ou négatives doivent être codées dans la base de faits. DTF utilise "l'hypothèse du monde ouvert", qui affirme donc que nous ne disposons que d'une connaissance incomplète sur le monde qui nous entoure, et ceci pour la desscription sémantique (i.e., interprétation de la logique). Par conséquent, les informations manquantes ou les informations que nous ne pouvons pas déduire de la base sont supposées inconnues. Si ϕ est un fait, $\neg\phi$ est vraie (i.e., prouvée) s'il peut être déduit à partir de la base de connaissances. Comme un contexte est une séquence de faits, il en découle que DTF n'admet que les preuves constructives. Il est à noter que au niveau de la syntaxe le raisonnement ne peut porter que sur un système complet conduisant à assumer l'hypothèse du monde clos au niveau de la syntaxe.

2.3.2 Expression des constantes

Le type unité (qui permet de représenter les constantes), dénoté par 1, est un type qui ne contient qu'un seul élément, noté $*$ comme preuve. Les règles de formation et d'introduction correspondantes sont :

$$\frac{\Gamma\ \ valid}{\Gamma \vdash 1 : Type}\ (1 - \mathsf{form}) \qquad \frac{\Gamma\ \ valid}{\Gamma \vdash * : 1}\ (1 - \mathsf{intro})$$

Avec $C[z]$ qui représente le produit dépendant $\Pi z : 1 \,.\, C(z)$, la règle d'élimination s'écrit :

$$\frac{\Gamma \vdash C : 1 \to T \;\; \Gamma,\; z : 1 \;\; \vdash c : C[*/z]}{\Gamma, x : 1 \vdash \phi(C, c)(x) : C[x]} \; (1 - \mathsf{elim})$$

Avec la règle de calcul suivante :

$$\phi(C, c)(*) \quad \rightsquigarrow \quad c$$

Dans ces règles, T est un type et ϕ une fonction qui applique à un type de donnée (C) à une son terme constant (c) vers sa valeur, i.e., c. Par exemple, prenons C un type constant représentant une distance en mètre et supposont que nous voulions représenter la valeur constante 2 mètres. Avec les règles précédentes, nous pouvons écrire $C : 1 \to \mathit{distance}$ et $c = 2$, qui s'interprète par c est une valeur du type C (à une seule valeur) dont la syntaxe est $\mathit{distance}(2)$. Alors, la règle de calcul du type unité nous permet de dire que $\phi(1 \to \mathit{distance}, 2)(*)$ est prouvée par une seule valeur, c'est-à-dire, 2. Remarquons que cette règle peut être appliquée dans la définition d'une structure de donnée, par exemple, la valeur maximum d'une distance qui serait égale à 2 peut être comparée dans un *Sigma*-type à une valeur constante de la manière suivante :

$$\sigma \;\triangleq\; \Sigma\, d : \mathit{distance} \,.\, \Sigma\, d_m : \mathit{distance}(2) \,.\, \mathit{plusPetitQue}(d, d_m)$$

Ce qui veut dire que d_m est du type $\mathit{distance}(2)$. Qui plus est, nous avons $\mathit{distance}(2) \leq \mathit{distance}$, alors il peut être converti selon la règle (Sub) en un type correct.

2.4 Le noyau de DTF

2.4.1 Les $\Sigma - \mathit{types}$ emboîtés comme représentation des contextes

Les contextes sont reliés par une relation de partonomie qui forme ainsi une relation d'ordre partiel entre ces contextes. Celle-ci peut être représentée par des Σ-types emboîtés. La relation de sous-typage entre les contextes est implémentée directement par le fait qu'un type de contexte est utilisé dans la définition d'un autre type contexte, deux contextes ne pouvant cependant pas être contenus au même niveau de définition d'un contexte.

Définition 28. Le type d'un contexte $\mathcal{C}$ est exprimé par une somme de Σ-types dont les éléments sont des concepts et des propositions. Les Σ-types non-dépendants et emboîtés sont présents dans une même structure.

$$\begin{aligned}
\mathcal{C} \;::=\;& \Sigma\, \mathcal{V}_1 : T_1 \,.\, \Sigma \mathcal{V}_2 : T_2 \,.\, \ldots \,.\, \Sigma \mathcal{V}_n : T_n \,.\, P(\pi_{1_1} \ldots \pi_{k_1} \mathcal{V}_1, \ldots, \pi_{1_n} \ldots \pi_{k_n} \mathcal{V}_n) \\
\mid\;& \Sigma\, \mathcal{V}_1 : T_1 \,.\, \Sigma \mathcal{V}_2 : T_2 \,.\, \ldots \,.\, \Sigma \mathcal{V}_n : T_n \,.\, \Sigma \mathcal{V}_{n+1} : \mathcal{C} \,.\, P(\pi_{1_1} \ldots \pi_{k_1} \mathcal{V}_1, \ldots, \pi_{1_{n+1}} \ldots \pi_{k_{n+1}} \mathcal{V}_{n+1}) \\
\mid\;& \mathcal{C} \times \mathcal{C}
\end{aligned}$$

où $n \geq 1$, T_i sont des types et P, une proposition. Si T_i est un type somme, alors il peut être emboîté dans k_i niveaux où chaque π_{k_1} sont valides pour π_1 ou π_2. Si T_i est un type de base, alors $\pi_{1_i} \ldots \pi_{k_i} \mathcal{V}_i = \mathcal{V}_i$. Notons que tous les contextes types sont des types somme, mais que tous les types somme ne sont pas des contextes.

Les $\Sigma - types$ emboîtés peuvent décrire un type de contexte pour lequel les variables liées représentent la connaissance physique nécessaire à la modélisation des concepts, des propriétés et des contraintes. Les structures de contextes peuvent décrire des entités simples du contexte comme la variable d'une équation logique jusqu'aux contextes complexes représentant des moments de la vie réelle. Ce résultat correspond au principe dit de localité [65], affirmant qu'une information valable pour un contexte donné est une fonction des objets présents dans le contexte.

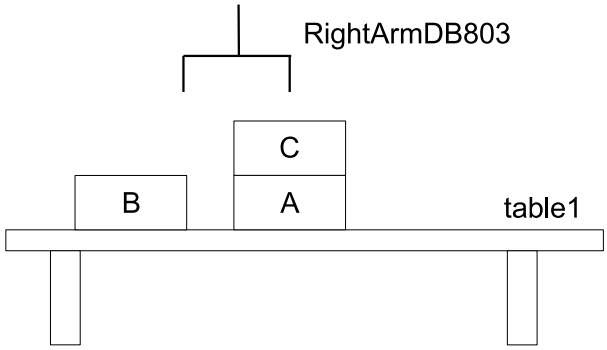


FIGURE 2.2 – Exemple : le monde des blocs.

Voici, pour illustrer cela, l'exemple du monde des blocs ("block world system") pour lequel les prédicats sont les suivants :

$$
\begin{aligned}
T_1 &\triangleq \Sigma x : block.\Sigma y : table.On(x,y) \\
T_2 &\triangleq \Sigma x : block.\Sigma y : block.On(x,y) \\
T_3 &\triangleq \Sigma x : block.Clear(x) \\
T_4 &\triangleq \Sigma x : RobotArm.empty(x) \\
T_5 &\triangleq \Sigma x : RobotArm.\Sigma y : block.Hold(x,y)
\end{aligned}
$$

Ces prédicats caractérisent la position relative des blocs et de l'état de la main du robot. La situation courante est donnée par les preuves :

$$
\begin{aligned}
& A, B, C \ : \ bloc \\
& table1 \ : \ table \\
& RightArmDB803 \ : \ RobotArm \\
q_1 \quad & est\ une\ preuve\ de\ On(A, table1) \\
q_2 \quad & est\ une\ preuve\ de\ On(B, table1) \\
q_3 \quad & est\ une\ preuve\ de\ On(C, A) \\
q_4 \quad & est\ une\ preuve\ de\ Clear(B) \\
q_5 \quad & est\ une\ preuve\ de\ clear(C) \\
q_6 \quad & est\ une\ preuve\ de\ Empty(RightArmDB803)
\end{aligned}
$$

Si une action (e.g. Take()) nécessite un contexte pour lequel au moins un bloc repose sur la table, la main du robot est vide et au moins un bloc n'a pas de bloc au dessus de lui, alors le contexte peut être décrit par le $\Sigma - type$ suivant :

$$C_0 = T_3 \times T_4 \times \Sigma\, w\, :\, T_1\,.\,Clear(\pi_1 w)$$

Pour un T_3, deux preuves sont obtenues, $< B, q_4 >$ et $< C, q_5 >$, alors que pour T_4, la preuve de la vérification est $< RightArmDB803, q_6 >$. Il existe également deux preuves pour le T_1, $< A, < table1, q_1 >>$ et $< B, < table1, q_2 >>$ mais comme le type dépendant est contraint par le membre de gauche de ces paires qui peuvent être au choix B ou C (avec pour preuves q_4 et q_5), alors la preuve du second terme est donnée par C_0, $<< B, < table1, q_2 >>, q_4 >$. Finalement, les deux preuves de C_0 s'écrivent :

$$< B, q_4 >< RightArmDB803, q_6 >< < B, < table1, q_2 >>, q_4 >$$
$$< C, q_5 >< RightArmDB803, q_6 >< < B, < table1, q_2 >>, q_4 >$$

Comme il existe deux preuves, il est possible de choisir entre deux possibilités pour le robot i.e., soit le bloc B, soit le bloc C, évitant ainsi les conflits (la décision dépendant de l'objectif à atteindre).

2.4.2 La représentation des contextes avec DTF

Pour pouvoir expliquer le choix des structures utilisées pour représenter les contextes en DTF, il convient de fournir quelques explications concernant les relations pouvant exister entre les situations et les contextes.

Représentation des contextes

Généralement, une situation représente un instantané de l'état du monde à un instant précis. Cette représentation dynamique formée d'une collection d'entités n'est associée à aucune structure représentative. L'observation de ces entités et de leurs relations permet de constater que les objets physiques peuvent être conceptualisés par une ontologie (généralement une ontologie de domaine), alors que les représentations des contextes sont déduites d'une analyse résultant d'informations fournies par exemple par des capteurs symboliques [13].

Un contexte est une structure dépendante composée de concepts et de contraintes. Les concepts vérifiés de la situation en cours forment des ensembles dont la preuve est donnée par les objets qui sont une instance de ces concepts dans une situation courante. Les contextes sont ainsi des sous-ensembles d'une partie de la situation ou de toute la situation. En partant de cette constatation, nous étudierons comment les Σ-types emboîtés apparaissent comme la représentation la plus adaptée pour typer les contextes. Avant de continuer, il faut préciser qu'il existe une autre structure qui fut analysée, ce sont les enregistrements à types dépendants.

La subsomption et la relation partonomique

Comme la fait remarquer McCarthy propose une hiérarchisation pour résoudre des problèmes en intelligence artificielle il est nécessaire de pouvoir raisonner, non seulement, sur des contextes, mais également sur les contextes dont un contexte est une partie prenante. Cet aspect peut être mis en oeuvre grâce à la relation de subsomption et la relation partonomique [18]. Avec la relation partonomique les contextes sont construits d'une manière incrémentale à partir du contexte vide

18. Aussi appelée relation « Partie - Tout ».

(racine) et cela correspond à ce qui est appelé par McCarthy un "context lifting". La plupart des modèles en représentation des connaissances nécessitent deux opérations : la subsomption (i.e., des structures qui héritent des propriétés de structures existentes) et l'agrégation [19](i.e., deux structures différentes peuvent être réunies en une seule). En fait, ces opérations peuvent être aisément décrites par un mécanisme reposant sur le type somme. La subsomption correspond au sous-typage entre des types de base, alors que la relation partonomique s'exerce entre des types somme.

Prenons un ensemble des types de base, définis par la grammaire $\mathcal{T} ::= \mathcal{V}$, où $\mathcal{V}$ est l'ensemble des types des variables et des constantes. Alors, la règle (Sub) est vérifiée pour les types de base. Par exemple, si l'on considère les types capteur et périphérique, nous pouvons affirmer que capteur $\leq$ périphérique indiquant qu'un capteur possède une information plus précise qu'un périphérique.

Définition 29. Pour les types de base la notion de sous-typage représente la subsomption.

Les relations entre les concepts sont décrites par le type somme, il est donc nécessaire d'expliquer comment il est possible de représenter la relation partonomique entre des types somme. Pour ces types, elle est exprimée par un mécanisme indiquant qu'un type T' est une partie d'un autre type T si T' contient plus d'informations que T. Il nous faut pour cela introduire des structures emboîtées qui sont des types somme composés.

Définition 30. (Type somme emboîté) Etant donné un type somme $\Sigma x : A.B$, soit M et M' des variables telles que $M : \Sigma x : A.B$ et $M' : \Sigma z : (\Sigma x : A.B).B'$, alors le type $\Sigma z : (\Sigma x : A.B).B'$ est appelé un type somme emboîté.

En d'autres mots, un type emboîté est un type somme contenant un autre type somme comme argument.

Définition 31. (Contextes partonomique) Un contexte type C' est une partie d'un autre contexte type C si C' contient au moins tout les types vérifiés dans C et si les types communs sont dans une relation de sous-typage selon le corollaire 2.

L'agrégation doit être prise en compte en considérant qu'un tout est divisé en parties par un mécanisme inverse à celui de la relation « Partie - Tout ». Comme les parties C et C' sont indépendantes, leur agrégation correspond au produit cartésien $C \times C$. Dans ce cas, un nouveau type de contexte peut être formé par l'agrégation de deux types de contexte qui forme un tout.

Définition 32. (Agrégation de contexte) L'agrégation d'un contexte C avec un autre type de contexte C' correspond au produit des contextes non dépendant $C \times C'$.

D'un point de ontologique, ce résultat correspond à la relation ontologique « a pour partie » qui est, la relation inverse de « Partie - de »).

Théorème 4. Si un type de contexte C est une partie de C', alors la règle suivante est vérifiée :

$$\Gamma \vdash \Pi x_1 : T_1 \,.\, \ldots \,.\, \Pi x_m : T_m . \ldots \,.\, \Pi x_n : T_n . C(x_1, ..., x_m, ..., x_n) \rightarrow C'(x_1, \ldots, x_m) \qquad (2.4)$$

Preuve Nous devons distinguer deux cas. Dans le premier cas un type de contexte dépend d'un autre type de contexte auquel des informations sont ajoutées et dans le deuxième elle concerne un type de contexte composé de types de contexte indépendants. Un exemple générique du premier cas est fourni par le type somme emboîté. En applicant la règle d'introduction ($\Sigma -$

19. Relation inverse de la relation « Partie - Tout ».

intro) dans l'environnement Γ, il s'ensuit que $a : A$ et $b : B[a/x]$, alors $\Sigma x : A.B$ est habité par $\langle a, b\rangle$. En appliquant à nouveau la règle ($\Sigma -$ intro) pour $\Sigma z : (\Sigma x : A.B).B'$ pour toutes paires $\langle a, b\rangle : \Sigma x : A.B$ et $b' : B'[\langle a, b\rangle/z]$ nous obtenons la preuve $\langle\langle a, b\rangle, b'\rangle$. Alors, pour chaque objet du type $\Sigma z : (\Sigma x : A.B).B'$ nous obtenons l'objet du type correspondant $\Sigma x : A.B$. Cette démonstration se généralise par induction pour un nombre d'emboîtement quelconque.

Pour le type somme indépendant, e.g., avec $\Sigma x : A.B$ habité par $\langle a, b\rangle$ et $\Sigma x : A'.B'$ habité par $\langle a', b'\rangle$, il est aisé de voir que n'importe quelle preuve de $\Sigma x : A.B \times \Sigma x : A'.B'$, i.e., $\langle\langle a, b\rangle, \langle a', b'\rangle\rangle$ est (extensionnellement) une preuve de $\langle a, b\rangle$ (ou $\langle a', b'\rangle$). □

Théorème 5. La subsomption et la relation partonomique entre les contextes sont dérivables dans DTF.

Preuve La grammaire des types de contextes montre qu'ils sont composés soit par des types sommes emboîtés, soit construits par produits non-dépendants de type somme. Dans les deux cas, en appliquant le théorème 4 et le corollaire 2, on en déduit que la relation partonomique entre contexte est dérivable des axiomes de DTF. □

2.4.3 Les enregistrements à types dépendants

Les enregistrements à types dépendants sont une structure de données intéressante qui pourrait être employée pour décrire des contextes. En voici une courte introduction.

Notion de labels d'enregistrement. A la différence des variables, les labels sont des accesseurs globaux d'enregistrement qui sont non alpha convertibles.

Notion de champ. Un champ est une paire dont le premier élément est un label indexé l_i et le second est un objet dans le cas d'un enregistrement, et un type dans le cas d'un type d'enregistrement.

Il est supposé dans le reste du texte que les majuscules dénotent des types et les minuscules, les objets - ou tokens -. Pour la théorie constructive des types, l'introduction des enregistrements à types dépendants (DTR) [14] est une extension du framework de Martin-Löf avec des DRT et les sous-typages entre enregistrements.

La première formalisation des enregistrements à types dépendants se trouve dans le travail de Harper et Lillibridge [73]. Pour Robert Pollack [113], il existe deux types d'enregistrements à types dépendants, les enregistrements associés à droite et les enregistrements associés à gauche, alors que pour Alexei Kopylov [84] ce ne sont que deux chemins différents pour construire le même type.

Pour représenter par exemple le contexte d'une tâche de mise en route d'un lecteur DVD neuf relié à un téléviseur et sans aucune image apparaissant sur le téléviseur, il est possible d'utiliser les Σ-types de la manière suivante :

$$\Sigma\, x \,:\, lecteurDVD\,.\,\Sigma p_1 \,:\, neuf(x)\,.\,\Sigma y \,:\, TV\,.\,\Sigma\, p_2 \,:\, connexion(x, y)\,.$$
$$\Sigma\, z \,:\, image\,.\,\Sigma\, p_3 \,:\, absente\_de(z, y)$$

En ce qui concerne les objets des Σ-types, prenons q_1 une preuve de *neuf*, q_2 une preuve de *connexion* et q_3 une preuve de *absente_de* :

$$< TangentD102, < q_1, < SonyTrinitron, < q_2, < image\_t, q_3 >>>>>$$

L'idée est donc d'introduire des enregistrements pour remplacer les variables liées (p.ex., x, y et z) par des labels afin d'obtenir une structure plus lisible et plus compacte.

Certains constructeurs des DRT les rapprochent des Σ-types selon le choix de représentation (i.e. ajout à droite ou à gauche). Cependant, si elles offrent l'avantage d'une représentation plus "lisible", les règles de construction et d'élimination sont sujettes à discussion et beaucoup d'inconnues subsistent [113]. Aucun travail d'incorporation de cette primitive à ECC n'existe à ce jour. Ainsi, elle n'a pas été retenue pour ce travail.

2.4.4 La représentation des types d'actions

Un système intelligent prévu pour être appliqué dans des réseaux qui s'adaptent aux contextes (context-aware) peut utiliser les connaissances contenues dans une base de données pour produire des actions proactivement selon le contexte courant. Par ailleurs, le concept de contexte est un moment universel. Il nécessite donc l'ajout d'un concept intentionnel de la manière suivante :

Définition 33. Etant donné un concept intentionnel, un contexte est le minimum d'informations qui contraignent la sémantique d'une action.

Les concepts intentionnels sont les actions, le diagnostic, etc. Par exemple, en considérant un contexte relié à une action donnée, la connaissance précise décrite dans le contexte donne le sens précis de l'action. En considérant le contexte d'une application, qui est le contexte de l'action, l'application peut par exemple être divisée en séquences d'actions. En outre, comme l'approche est constructive, les actions ne sont pas mappées dans la représentation du monde ou dans un système de mondes possibles, mais plutôt représentées par la connaissance pratique des agents.

Dans [87], l'auteur montre qu'une action peut être décrite par un type. Alors, comment représenter le fait qu'une action est une intention d'un contexte donné ? Un jugement critique est nécessaire pour pouvoir décider si une certaine procédure peut être ou non l'intention d'un contexte donné. Par conséquent, une structure canonique est nécessaire. Après avoir choisi une représentation des contextes par des Σ-types, il apparaît intéressant d'adopter la même structure pour la représentation de l'expression des liens entre un contexte et son intention. En effet, plutôt que de choisir une représentation par des fonctions, cette structure offre l'avantage de traduire l'aspect relationnel du lien entre les contextes et les actions. Ainsi l'action n'est pas une conséquence d'un contexte, mais une association possible entre les deux, laissant la possibilité de relier d'autres contextes à une action et de permettre une gestion éventuelle de l'historique des paires valides.

Définition 34. Etant donné un type C dépendant des types $T_1, \ldots, T_n$, un type d'action (contextualisé) est décrit par un prédicat qui fait intervenir m types ($m \leq n$), dont le type est $T_1 \rightarrow \ldots \rightarrow T_m \rightarrow Prop$. La relation entre les contextes et les actions est formalisée par le Σ-type suivant :

$$\Sigma\, c : C\ .\ A(\pi_{1_1} \ldots \pi_{k_1} c, \ldots, \pi_{1_m} \ldots \pi_{k_m} c)$$

dans lequel la variable c est un contexte, A est un prédicat d'action, k représente le niveau de la projection approprié et chaque π_{i_j} vérifié pour π_1 or π_2.

L'action A est disponible car elle est liée à un contexte valide. Par exemple avec le monde des blocs (blocks-world), l'action contextualisée $\Sigma c : C_0.Take(\pi_1\pi_1\pi_2 c, \pi_1\pi_1\pi_2\pi_2 c)$ est prouvée par :

$$\langle\langle C, q_4\rangle\langle RightArmDB803, q_6\rangle\langle B, table1, q_2, q_4\rangle, Take(RightArmDB803, B)\rangle$$

où les deux arguments de $Take$ sont respectivement du type $RobotArm$ et $block$, et ils sont utilisés par le type action : $Take : RobotArm \rightarrow block \rightarrow Prop$.

Ici, le Σ-type C_O est le contexte de l'action qui consiste à attraper un bloc si le bras du robot est libre. Une action est prouvée si le système a une fonction effective (décrite dans un langage de programmation quelconque) capable d'effectuer cette fonction. Dans l'exemple précédent, il est supposé qu'il existe une fonction capable de contrôler le bras d'un robot. Cette association contexte + action doit être vue comme une action exécutée dans un contexte particulier qui donnera des effets particuliers.

Les types sont extraits et vérifiés depuis une ontologie locale et correspondent sémantiquement à ce que Giunchiglia appelle le "principe de localité". En utilisant la théorie des types [87], la fin d'une action a (i.e., les effets) est représentée par un type $Prop$ et le résultat est $E : Prop$. La connaissance du but à obtenir ($E : Prop$) et du moyen d'arriver au résultat ($a : A$) permet de transformer une connaissance pratique en action. Si E est un effet, alors E est réalisable s'il existe un objet du type de E. Par exemple, dans le cas précédent, l'effet ajoute une preuve de σ_5. Comme $x : RobotArm$ est prouvé et pour $y : block$, le contexte q_7 est prouvé par la proposition $Hold(RightArmDB803, C)$ et aussi pour le Σ-type (e.g., $< RightArmDB803, < C, q_7 >>$).

2.4.5 Les axiomes de bases

Le cadriciel DTF est un langage pour la spécification et le raisonnement sur les systèmes dynamiques. Ce langage est constitué par trois ensembles non vides, les contextes, les contextes agrégés et les actions. Avec DTF les représentations partielles du monde réel s'expriment en termes de collections de preuves relatives à une situation particulière. Etant donné une situation S et l'ensemble $\mathcal{C}$ des types de contextes, les axiomes suivants sont vérifiés :

A1 Etant donné une structure C décrivant un contexte, alors C est vérifié dans S ssi tous les termes qui sont présents dans C sont instanciés avec l'ensemble des faits F_C tel que $F_C \subseteq S$.

A2 Un ensemble fini de contextes (éventuellement vide) peut être valide dans S.

A3 Une situation est consistante si $S \neq \{\emptyset\}$.

A4 Aucune structure de contexte ne doit prouver de contradiction sémantique.

A5 $\forall A \in \mathcal{A}$ où A est un nom d'action, alors $\exists\, C \in \mathcal{C}$ avec F_C l'ensemble des termes instanciés de C tel qu'il $\exists\, x_1 \in F_C, ..., x_k \in F_C \mid x_1 \rightarrow x_2 \ldots \rightarrow x_k \rightarrow A$, où C est une structure de contexte , $\rightarrow$ dénote le symbole de la fonction usuelle et $x_1, ..., x_k$ sont des termes de C.

A6 $\forall A, A' \in \mathcal{A}$ où A, A' sont des noms d'actions, si $\exists\, C \in \mathcal{C}$ avec F_C l'ensemble des termes instanciés de C tel que $\exists x_1 \in F_C, ..., x_k \in F_C \mid x_1 \rightarrow x_2 \ldots \rightarrow x_k \rightarrow A$ et $x_1 \rightarrow x_2 \ldots \rightarrow x_k \rightarrow A'$, alors $A = A'$, avec la même syntaxe que pour l'axiome 5.

Il faut noter que l'axiome **A6** est un principe raisonnable pour une conception objective.

2.4.6 La décidabilité

Définition 35. Etant données les deux paires (contexte, action) : $\Sigma\, c : C\,.\,A$ et $\Sigma\, c' : C'\,.\,A'$ tels que C et C' ne sont pas dans une relation de partonomie, alors $C'' = C \times C'$ est un type de contexte agrégé ssi :

$$\exists\, A \,|\, \Sigma\, c : (C \times C')\,.\,A$$

Proposition 6. Etant donné un type de contexte agrégé $C \times C'$, alors $C \leq C \times C'$ et $C' \leq C \times C'$ sont vérifiées.

Preuve : évidente.

□

La relation de partonomie entre les contextes consiste à ajouter de l'information à un contexte sans que celle-ci corresponde elle-même à un contexte (ceci exclut donc les contextes agrégés).

Théorème 7. La relation de partonomie entre les contextes est un arbre de subsomption, excluant les contextes agrégés (graphe connexe sans cycle).

Preuve : la relation $\leq$ est une relation partielle, réflexive, transitive et antisymétrique , donc un ordre partiel. La relation $\leq$ forme une unique hiérarchie contenant tous les types avec un type racine unique. Il n'existe pas de cycles dans cette structure. Les contextes types sont donc les noeuds d'un arbre ordonné par la relation de partonomie.

□

La relation de composition entre les contextes $(C \times C')$ consiste à définir un nouveau contexte formé de l'union des contextes et correspondant à une action. La recherche de contextes consistera à parcourir chaque branche de l'arbre tant que les contextes peuvent être prouvés. Puis à effectuer une recherche sur les contextes prouvés les plus profonds dans chaque branche de l'arbre précédant, obtenant ainsi une forêt[20]. Par exemple, les noeuds fuschia de la figure 2.3 représentent la forêt des contextes prouvés.

Théorème 8. (Décidabilité de la recherche de contexte)
Il existe toujours au moins une paire (contexte, action) dans une situation donnée.

Preuve :

1. Dans le cas où aucun contexte n'est vérifié, seule la paire $<<>, \mathit{stop}>$ est valide et le programme s'arrête (action stop).
2. Dans tout autre cas, comme les contextes forment un arbre (cf. théorème 7) cela signifie qu'il existe au moins un contexte valide, et donc une action à entreprendre.

□

Il est possible de trouver plusieurs contextes valides dans une situation donnée :
– Si ces contextes sont dans une relation de partonomie, alors le contexte le plus précis est choisi.

20. Un arbre est un graphe connexe sans cycle. Un graphe sans cycle qui n'est pas connexe est appelé une forêt (chaque composante connexe est un arbre).

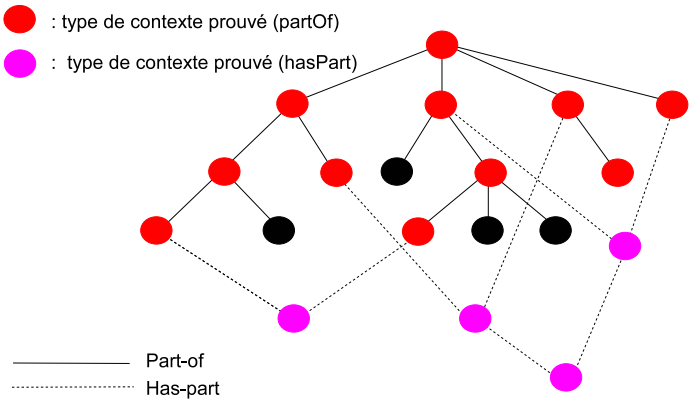


FIGURE 2.3 – Exemple de forêt de contextes prouvés

– Si les contextes sont agrégés, alors les actions associées aux contextes agrégés valides sont sélectionnées.

Remarque : dans le cas où il est nécessaire d'exécuter une action unique, il est possible d'utiliser DTF pour trouver une solution à travers la notion de spécification, mais ceci reste encore à explorer.

2.4.7 L'expressivité

Pour mesurer l'expressivité d'un langage logique, il est fondamentale de pouvoir le comparer à d'autres langages. La mesure qualitative qui est suggérée ici est faite par rapport à une logique de description (DL) possédant un nombre minimal d'opérateurs.

Théorème 9. ([80]) La théorie des types dépendants (au sens de Bart Jacobs) peut être traduite dans la logique d'ordre supérieur (HOL).

Corollaire 10. DTF est au moins aussi expressive que HOL.

Preuve DTF est un sous-système de UTT [92] qui possède une logique interne d'ordre supérieur. Alors, suivant le théorème 9 DTF peut être traduit dans HOL. Il en résulte que DTF est au moins aussi expressive que HOL. □

Soit un langage $\mathcal{L}^3$, le sous ensemble du calcul des prédicats du premier ordre avec des prédicats limités à trois arguments (triadique) et $\mathcal{L}^2$ le langage limité à deux arguments (dyadique). Une DL est définie par une fonction de traduction qui met en correspondance des concepts et des formules avec des variables libres x d'une part, et d'autre part les rôles et les formules avec des variables libres x et y.

Théorème 11. ([17]) Le langage de description avec des constructeurs de concepts $\{\top, \bot, \sqcap, \neg, \exists, fills, one-of\}$ et les constructeurs de rôles $\{role-and, role-not, product, inver\}$ est aussi expressif que le langage $\mathcal{L}^2$.

Théorème 12. DTF est plus expressif qu'une DL qui possède au moins les constructeurs de concepts $\{\top, \bot, \sqcap, \neg, \exists, fills, one-of\}$ et les constructeurs de rôles $\{role-and, role-not, product, inver\}$.

Preuve Du théorème 11, il s'en suit que toutes les DL qui ont au moins les opérateurs énumérés dans ce théorème sont au maximum aussi expressives que $\mathcal{L}^2$, qui est un sous-ensemble de la logique du premier ordre. Comme HOL est strictement plus expressive que la logique du premier ordre, nous pouvons en déduire que HOL est plus expressive qu'une DL qui possède au moins les opérateurs du théorème 11. Pour conclure, du théorème 9 nous pouvons dériver directement que DTF est plus expressif qu'une DL qui possède les opérateurs précédents. □

2.4.8 La représentation des propriétés ontologiques

De par son importance dans les représentations des relations ontologiques, la relation de tout et partie doit être intégrée dans le formalisme DTF. Pour cela, il est tout d'abord nécessaire d'expliquer comment des propriétés de la relation, par exemple la transitivité et la distributivité, peuvent être formalisées. Cette caractérisation des structures primitives se situe au niveau ontologique. L'intérêt ne porte que sur la relation de tout et partie pour laquelle la transitivité est vérifiée, puisqu'il n'existe pas de consensus concernant les autres relations. Notons que la théorie des types dépendants permet d'exprimer la transitivité comme une propriété qui dépend d'une valeur correspondant aux différentes manières par lesquelles les différentes parties contribuent à la structure du tout (e.g., les distinctions proposées par Winston, Chaffin and Herrmann [141]). Cependant, cet aspect ne sera pas développé ici.

DTF permet d'exprimer conjointement des relations quelconques (structure de données) ainsi que les propriétés et contraintes que doit vérifier cette structure au sein du même cadre formel. De cette manière, le contenu computationnel est "séparé" des preuves et de l'expression des contraintes.

Définition 36. (Spécification) Une spécification Sp est une paire contenant un type de Sp-structures $Struc[Sp]$ appellé structure type de Sp et un prédicat $Ax[Sp]$ sur la structure $Struc[Sp]$. Elle est décrite par un Σ-type :

$$Sp \triangleq \Sigma\, Struc\ :\ Type\,.\,Ax(Struc) \tag{2.5}$$

avec $Ax : Struc \rightarrow Prop$

La structure Sp est dite consistante s'il existe une Sp-structure et si cette structure satisfait les axiomes de $Ax[Sp]$. Dans la suite est utilisée la notation simplifiée $\sum[x_1 : T_1, x_2 : T_2, ..., x_n : T_n]$ pour dénoter $\Sigma x_1 : T_1.\Sigma x_2 : T_2. \ldots, .T_n$.

Toute relation est formalisée par le Σ-type :

$$Rel \triangleq \Sigma\, x\ :\ Type\,.\,\Sigma\, y\ :\ Type\,.\,R(x,y) \tag{2.6}$$

Transitivité

Toute relation transitive est définie par la spécification suivante :

$$Struc\,[Tr] \triangleq \sum \begin{bmatrix} Tr & : Rel \\ Transitive & : Rel \rightarrow Prop \end{bmatrix}$$

et pour toute relation r de type $Struc[Tr]$ (avec Tr une notation abrégée pour $Tr[r]$) :

$$\begin{array}{rl} & \forall u, u' : Tr\ . \\ Ax[Tr] \triangleq & (R_u = R_{u'} : Prop\ \ \&\ \ \pi_1\pi_2 u = \pi_1 u' : Type) \supset \\ & (R_u(\pi_1 u, \pi_1\pi_2 u)\ \ \&\ \ R_{u'}(\pi_1 u', \pi_1\pi_2 u') \supset R_u(\pi_1 u, \pi_1\pi_2 u')) \end{array}$$

où $R_u \triangleq \pi_2\pi_2 u$ et $R_{u'} \triangleq \pi_2\pi_2 u'$. L'axiome établit que si la proposition dans la structure Rel est identique ($R_u = R_{u'}$) et que si nous appliquons deux fois la relation avec le second argument de la première et que celui ci est identique au premier argument de la seconde, alors la relation est valide entre le premier argument de R_u et le second argument de $R_{u'}$. En clair, une preuve de la transitivité d'une relation R peut être obtenue en lisant cette information dans une table, ce qui prouve la structure $Struc[Tr]$, et toute relation de ce type doit satisfaire les axiomes $Ax[Tr]$ pour que la spécification soit vérifiée. Un des intérêts de ce mécanisme de spécification est qu'une spécification donnée peut être étendue et ré-utilisée dans d'autres spécifications. Cet aspect est crucial pour l'application des spécifications aux ontologies. Comme les relations méréologiques et topologiques sont à la base d'ontologies formelles, une spécification peut être illustrée dans le cas de la relation spatiale (ou temporelle) partie-de (partOf), qui est transitive.

Une preuve de $Struc[Tr](partOf)$ est donnée par vérification de la paire $< partOf, q_1 >$ avec q_1 une preuve de $Transitive(partOf)$. Plaçons-nous dans un cas où la relation $partOf$ est transitive, et en supposant qu'existent dans une base de connaissances, les termes u et u' :

$$\begin{array}{rcl} u & : & \Sigma x : soldier.\Sigma y : section.\ partOf(x, y) \\ u' & : & \Sigma x : section.\Sigma y : platoon.\ partOf(x, y) \end{array}$$

Il est supposé qu'il existe dans la base de données, les preuves respectives pour les deux relations : $< Paul, < sec35, p_1 >>$ et $< sec35, < P8, p_2 >>$ avec p_1 et p_2 les preuves respectives de $partOf(Paul, sec35)$ et $partOf(sec35, P8)$. Dans l'axiome $Ax[Tr](partOf)$, les conditions sont vérifiées ($R_u = R_{u'} = partOf$ et $\pi_1\pi_2\pi_1 u = \pi_1\sigma_1 u' = sec35 : section$) apportant ainsi la preuve de $partOf(Paul, P8)$ puisque nous avons simultanément $R_u(\pi_1\pi_1 u, \pi_1\pi_2\pi_1 u)$ (preuve $partOf(Paul, sec35)$ et $R_{u'}(\pi_1\pi_1 u', \pi_1\pi_2\pi_1 u')$ (preuve $partOf(sec35, P8)$).

Remarquons qu'au travers de la définition 2.6, la spécification de la transitivité ou de la distributivité s'applique à n'importe quel type. Si nous voulons nous restreindre à certains types, il suffit, soit de spécifier le type au niveau de la relation 2.6, soit de spécifier dans l'axiome un ou plusieurs prédicats que doivent satisfaire ces types.

Distributivité

Un autre cas intéressant est la distributivité à gauche et à droite [3]. Il ne sera considéré ici, faute de place, que la distributivité à gauche, la distributivité à droite pouvant se traiter de la même manière sans difficulté supplémentaire. Dans une collection d'agrégats d'individus appelée collection, la distributivité existe pour la relation $hasPart$ aussi appelée $hasItem$, $hasParticipant$, $hasElement$, etc. Toute relation DR distributive à gauche par rapport à une relation DR' est spécifiée par :

$$Struc[DR, DR'] \triangleq \sum \left[\begin{array}{ll} DR & : Rel \\ DR' & : Rel \\ L - Distrib & : Rel \rightarrow Rel \rightarrow Prop \end{array} \right.$$

et pour toute paire de relations r, r' de type $Struc[DR, DR']$ (avec DR, DR' une notation abrégée pour $DR[r], DR'[r']$) :

$$Ax[DR, DR'] \triangleq \begin{array}{l} \forall u : DR, \forall u' : DR' \ . \\ (\pi_2\pi_2 u = \pi_2\pi_2 u' \supset \bot \ \ \& \ \ \pi_1 u = \pi_1 u' : Type) \supset \\ (R_u(\pi_1\pi_1 u, \pi_1\pi_2 u) \ \ \& \ \ R_{u'}(\pi_1\pi_1 u', \pi_1\pi_2 u') \supset \\ R_u(\pi_1\pi_2 u', \pi_1\pi_2 u)) \end{array}$$

avec $R_u \triangleq \pi_2\pi_2 u$ et $R_{u'} \triangleq \pi_2\pi_2 u'$.

L'axiome établit que si les propositions des structures sont distinctes ($R_u = R_{u'} \supset \bot$) et que le premier argument de la première relation est identique au premier argument de la seconde, alors la relation de R_u est valide avec comme arguments respectifs, le premier argument de R_u et le second argument de $R_{u'}$. S'il existe une preuve de la distributivité d'une relation DR par rapport à R', c'est à dire une preuve de l'existence de $Struc[DR, DR']$, alors toute paire de relations de ce type doit satisfaire les axiomes $Ax[DR, DR']$ pour que la spécification soit vérifiée.

Les prédicats sur des collections s'appliquent de manière différente sur les articles composant la collection. Dans le cas, par exemple, de la distributivité à gauche interprétée pour une relation binaire générique qui possède une collection d'arguments à gauche, si une relation de type *hasPart* est distributive à gauche par rapport à une relation R, alors la relation R qui est valable pour un tout est aussi valable pour ses parties. Une règle similaire est également valable pour une relation distributive à droite.

Une preuve de $Struc[DR, DR'](hasObjective, hasPart)$ est donnée par vérification de la paire emboîtée $< hasObjective, < hasPart, q_1 >>$ avec q_1 une preuve de $L{-}Distrib(hasObjective, hasPart)$. Par exemple, le fait que les objectifs d'un groupe sont les mêmes que ceux de ses membres, peut s'exprimer par les types dépendants. Sachant que la relation *hasObjective* est distributive à gauche sur *hasPart*, où existent dans une base de connaissances, les termes u et u' tels que :

$$\begin{array}{rcl} u & : & \Sigma a : association.\Sigma b : topic \ . \ hasObjective(a, b) \\ u' & : & \Sigma x : association.\Sigma y : person \ . \ hasPart(x, y). \end{array}$$

alors, avec les preuves respectives établissant la différence des propositions et l'identification des premiers arguments, $(hasObjective = hasPart)$ est absurde et *association* est l'argument commun. Les preuves respectives de R_u et $R_{u'}$, e.g., $hasObjective(ACM - SIGART, AI)$ et $hasPart(ACM{-}SIGART, Patrick)$, permettent de déduire la preuve $hasObjective(Patrick, AI)$.

2.5 Etude de cas

Pour expliquer la théorie DTF développée à la section précédente, deux cas d'études sont proposés. Le premier décrit le fonctionnement d'un processus destiné à surveiller l'attribution de médicaments à des patients en milieu hospitalier. Le second est un système de gestion d'un agenda.

2.5.1 Le processus d'attribution de médicament

Le sujet d'étude est un exemple extrait de [59] dans lequel le problème consiste à montrer l'intérêt de l'utilisation des ontologies pour les réseaux adaptables au contexte. Dans le cas d'une assistance médicale aux patients, l'application peut être divisée en sous-processus comme par exemple un processus de supervision médicale qui traite des problèmes de la prescription de médicaments, de la surveillance et du renouvellement de traitement.

Dans un processus de décision sur les conditions médicales, i.e., les médicaments et leurs substitues, l'ensemble des assertions auxquelles doit correspondre le modèle sont par exemple :

i) Georges a pris froid.
ii) Georges utilise *Super Drug* (SD) un médicament à large spectre contre la grippe.
iii) SD lorsqu'il est utilisé contre la grippe se subsitue au sirop contre la toux CD.
iv) SD lorsqu'il est utilisé représente un risque si le malade a une maladie du foie.

La définition du modèle de l'application nécessite quatre concepts qui sont *Person*, *Drug*, Medical Condition (MC) et SD. Un médicament peut être utilisé pour traiter plusieurs maladies, et donc il peut posséder plusieurs substitues selon la maladie à soigner. Le modèle doit également refléter le fait qu'une personne est malade, qu'un médicament soigne une maladie et qu'une personne utilise un(des) médicament(s). Partant de ces assertions et d'une ontologie de domaine, les types dépendants suivants peuvent pour se soigner aisément les relations ontologiques exprimant les contextes :

$$\begin{aligned}
\sigma_1 &\triangleq \Sigma\, x : Person\,.\,(\Sigma\, y : Drug\,.\,usesDrug(x,y)) \\
\sigma_2 &\triangleq \Sigma\, s_1 : \sigma_1\,.\,\Sigma\, m : MC\,.\,forCondition(\pi_1\pi_2 s_1, m) \\
\sigma_3 &\triangleq \Pi\, s_2 : \sigma_2\,.\,hasCondition(\pi_1\pi_1 s_2, \pi_1\pi_2 s_2))
\end{aligned}$$

D'autre part, la règle de sous-typage suivante est ajoutée :

$$\Sigma\, x : Drug\,.\,\Sigma\, y : SuperDrug\,.\,x\, lessOrEqual\, y)$$

La première relation (σ_1) est vérifiée pour certaines personnes qui utilisent un médicament. La seconde dépend de la première et indique que connaissant le fait que des personnes utilisent des médicaments qui sont utilisés pour un traitement donné. L'inférence : σ_3 indique que si une personne utilise un médicament pour se soigner, alors cette personne se soigne contre une maladie traitée par ce médicament et, par voie de conséquence, a une des maladies couvertes par le médicament. Ici l'implication est réalisée par la quantification sur tous les individus qui possèdent la propriété σ_2, ce qui correspond à de la quantification paramétrique. Le Σ de σ_3 indique que le médicament peut être utilisé dans le cadre de plusieurs traitements puisque le résultat obtenu est un ensemble de preuves. Les relations sont indiquées à la figure 2.4. La relation $\leq$ est vérifiée

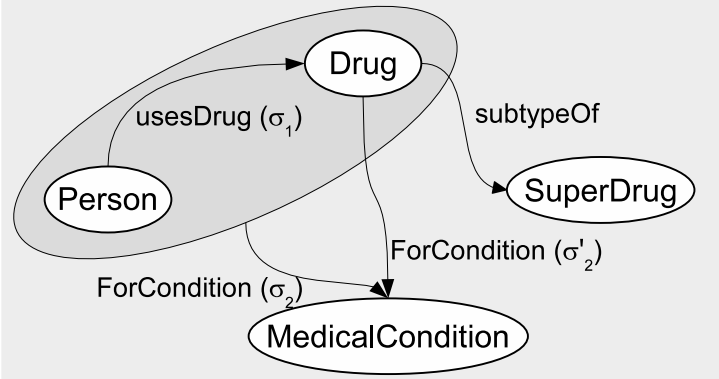


FIGURE 2.4 – Extrait de l'ontologie de domaine "traitement médical".

et si le concept de SD est plus général que le concept de $Drug$.

A la différence de la modélisation en OWL original qui utilise le produit cartésien $Drug \times MedicalCondition$, avec DTF il suffit pour exprimer les propriétés contenues dans le produit d'employer un Σ-type emboîté. Les auteurs mentionnent dans leur article les difficultés qu'ils ont rencontrées lorsqu'ils voulurent représenter en OWL les faits suivants :

a) Si une personne utilise un médicament, alors la personne a une des maladies traitées par le médicament.

b) Un médicament $D2$ qui se substitue à une autre drogue $D1$ lorsqu'il est utilisé pour soigner une maladie $C1$ est donc également un médicament pour cette maladie.

Les problèmes a) et b) sont directement résolus par la règle σ_3.

Le cas iii) est dérivé par les règles suivantes :

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash s_1 : \Sigma x : Person\,.\,\Sigma y : Drug \;.\; usesDrug(x,y)}{\Gamma \vdash \pi_2 s_1 : \Sigma y : Drug \;.\; usesDrug(\pi_1 s_1, y)}\,(\pi_2\text{-elim})}{\Gamma \vdash \pi_1\pi_2 s_1 : Drug}\,(\pi_1\text{-elim}) \quad \Gamma \vdash M : MC \quad \Gamma,\; x : Drug,\; y : MC \vdash N : ForCondition(\pi_1\pi_2 s_1/x, M/y)}{\Gamma \vdash < \pi_1\pi_2 s_1, < M, N >> \;:\; \Sigma x : Drug.\Sigma\, y \;:\; MC\,.\,forCondition(x,y)}\,(\Sigma\text{-intro})$$

Soit $s'_2 =< \pi_1\pi_2 s_1, < M, N >>$, la nouvelle relation suivante est obtenue

$$s'_2 : \Sigma x : Drug.\Sigma y : MC.forCondition(x,y)$$

qui relie les concepts $Drug$ et MC. Alors, par l'application de s'_2, le résultat est directement déduit pour obtenir toutes les preuves concernant un médicament D_1 utilisé pour un traitement donné $mc_1 < D1, < mc_1, q_1 >>$ avec q_1, une preuve de $forCondition(D1, mc_1)$. En utilisant une fonction de sous-typage $D1 \leq D2$, le résultat est $< D2, < mc_1, q_1 >>$ avec q_1, une preuve de $forCondition(D2, mc_1)$, qui indique que SD $D2$ lorsqu'elle est utilisée avec $mc_1 =$ "$cold$", est un substitut de $D_1 =$ "$CoughDrug$".

Finalement, il reste à prouver le fait que lorsqu'une SD est employée pour un traitement donné, elle peut représenter un risque en présence d'autres maladies dont le patient pourrait éventuellement souffrir (cas iv). En partant de l'hypothèse que le patient (e.g., Patrick) souffre d'une insuffisance hépatique et d'une grippe. Pour se soigner, il prend le médicament à spectre large $D2$ pour traiter sa grippe. Cependant, il existe un risque à utiliser ce médicament pour les personnes souffrant d'insuffisance hépatique. Cette situation est aisément décrite par les contextes C_1 et C_2 de la manière suivante :

$$\begin{aligned} \sigma_1 &= \Sigma x : Person \,.\, \Sigma y : Drug \,.\, usesDrug(x, y) \\ C_1 &= \Sigma t : \sigma_1 \,.\, \Sigma m : MC \,.\, hasCondition(\pi_1 t, m) \\ C_2 &= \Sigma c : C_1 \,.\, \Sigma mp : MedicalProblem \,.\, hasRiskWith(\pi_1\pi_2\pi_1 c, mp) \end{aligned}$$

en supposant maintenant que la situation valide les preuves

$$\left[\begin{array}{ll} x & = Patrick \\ y & = CoughDrug \\ q_1 & \\ m & = cold \\ q_2 & \end{array}\right.$$

avec q_1 une preuve de $usesDrug(Patrick, CoughDrug)$ et q_2 une preuve de $hasCondition(Patrick, cold)$. Ainsi, le contexte C_1 est prouvé et le diagnostic est indiqué par un message (e.g., message "ok").

Si dans la situation courante, le médicament $CoughDrug$ est remplacé par le médicament à large spectre D_2 alors, comme $CoughDrug$ est un sous-type de D_2 par sous-typage, le contexte C_1 est validé. Mais s'il apparaît que Patrick a une insuffisance hépatique, alors $mp = liverProblem$ et la base de connaissances permettent de prouver $q_3 = hasRiskwith(CoughDrug, liverProblem)$ ce qui valide le contexte C_2 et comme C_2 est un sous-type de C_1, alors C_2 est préféré [21] pour le diagnostic en cours et déclenche l'action associée, par exemple "warning". Dans une perspective pratique, une ontologie des médicaments peut résider sur un serveur centralisé, alors que la connaissance dynamique (e.g., sur Patrick) peut être contenue dans un assistant personnel, avec un raisonnement non-monotone sur les paires $\langle C, A \rangle$

2.5.2 La localisation spatio-temporelle d'une personne

Une personne gère son emploi du temps (e.g., un PDA) dans lequel il existe un certain nombre de réunions auxquelles elle doit participer. Chacune de ces réunions est indiquée par une heure de début est une heure de fin. La réunion a lieu dans une pièce donnée. Cette pièce est une partie d'un immeuble avec des coordonnées géographiques ou au moins une adresse. L'objectif est de déterminer la position d'un individu en utilisant les informations concernant ces réunions, plus précisément :

1. Déterminer la réunion à laquelle participe la personne.
2. Déterminer la localisation de la pièce où la réunion a lieu.

21. Comme C_2 contient une information plus précise que C_1.

3. Connaissant la localisation de la pièce dans laquelle la réunion a lieu, en inférer la localisation de la personne.

Une personne est dans une pièce où se déroule une réunion à laquelle elle doit assister. L'heure de début de la réunion est inférieure ou égale à l'heure actuelle, et l'heure de fin de la réunion est supérieure ou égale à l'heure actuelle. Il est difficile de modéliser des opérations de comparaisons et de définitions paramétriques entre des classes (il existe cependant des possibilités en introduisant des intervalles), alors qu'avec les types dépendants cela ne pose aucun problème. Un contexte peut être associé à cette réunion considérée comme un processus. Des briques de connaissances peuvent être facilement réalisées par les prédicats suivants :

$$\begin{aligned}\sigma_1 &= \Sigma t : Time\,.\,\Sigma t_s : Time(11/11/08 : 14.00)\,.\,GreaterThan(t, t_s)\\ \sigma_2 &= \Sigma s : \sigma_1\,.\,\Sigma t_e : Time(11/11/08 : 16.30)\,.\,LowerThan(\pi_1 s, t_e)\\ \sigma_3 &= \Sigma x : Person\,.\,\Sigma m : meeting\,.\,attends(x, m)\end{aligned}$$

Si l'intention (i.e. le processus de réunion) nécessite un contexte dans lequel l'heure courante est dans les limites de l'événement et concerne les personnes participants à l'événement, ce contexte peut être décrit par le $\Sigma - type$ suivant :

$$K_0 = \sigma_2 \times \sigma_3$$

L'heure de début et l'heure de fin de la réunion sont respectivement définies par deux valeurs constantes numériques. Les deux propositions (resp. plus grande que et plus petite que) sont toutes les deux prouvées, si et seulement si, l'heure courante $t : Time$ se situe entre l'heure de début et l'heure de fin. Il est à remarquer, que si σ_2 est prouvé, alors, la valeur de l'heure est dans les limites demandées. Toute réunion peut exploiter ce contexte à condition que les valeurs constantes soient adaptées en conséquence.

$$\Pi k : K_0\,.\,T\_part\_of(\pi_1\pi_2 k, \pi_1\pi_2\pi_2 k) \tag{2.7}$$

Le contexte précédent, K_0, peut être associé à une inférence montrant que toute occurrence de ce contexte implique que toute personne qui satisfait les contraintes du contexte est une partie temporelle de la réunion. Des contextes temporels peuvent être agrégés en ajoutant plus d'informations sur la réunion, telle que, par exemple, le sujet de la réunion, etc. Est-il possible en connaissant la position de la réunion d'inférer la position d'une personne ? Pour cela, une analyse des connaissances ontologiques est nécessaire. Premièrement, un type somme exprime la relation contextuelle [22] sur la pièce et la réunion :

$$\begin{aligned}\tau_1 &= \Sigma x : meeting\,.\,\Sigma y : room\,.\,hasLocation(x, y)\\ \tau_2 &= \Sigma x : meeting\,.\,\Sigma y : Person\,.\,hasParticipant(x, y)\end{aligned} \tag{2.8}$$

En imaginant qu'une réunion est une collection (temporelle) de personnes, et en supposant que dans la base de connaissances il existe une preuve de la structure $Struc[DR, DR']$

22. Le contexte de l'activité d'une personne.

($hasLocation, hasParticipant$), i.e. un cas particulier de $< hasLocation, < hasParticipant, r_1 >$ avec r_1 une preuve de $L\_distrib$ ($hasLocation$,$hasParticipant$) alors, la propriété de distributivité à gauche peut être appliquée à la relation $hasParticipant$. Par exemple, si les preuves de τ_1 et τ_2, sont respectivement $< Ontologies\ et\ le\ Web, < AulaB7, p_1 >>$, et $< Ontologies\ et\ le\ Web, < Richard, p_2 >>$ avec p_1 et p_2 les preuves respectives $hasLocation(Ontologies\ et\ le\ Web, AulaB7)$ et $hasParticipant(Ontologies\ et\ le\ Web, Richard)$. Alors, comme les propositions dans les structures sont différentes et que les premiers arguments dans les Σ-types τ_1 et τ_2 sont identiques, alors c'est une preuve de $hasLocation(Richard, AulaB7)$. Un processus identique peut être appliqué à la dernière inférence.

La base de connaissances peut inclure les relations suivantes :

$$\begin{aligned} \tau_3 &= \Sigma x : building\ .\ \Sigma y : address\ .\ hasLocation(x, y) \\ \tau_4 &= \Sigma x : building\ .\ \Sigma y : room\ .\ hasPart(x, y) \end{aligned} \tag{2.9}$$

Alors, en partant du principe qu'un immeuble est une collection de pièces (dans l'espace), et que les preuves suivantes sont vérifiées (e.g., preuves $< G{-}10, <$ "$B{-}1050, Brussel, Belgium$", $p_1 >>$ et $< G - 10, < AulaB7, p_2 >>$ où p_1 et p_2 sont des preuves respectives de $hasLocation(G - 10,$ "$B{-}1050, Brussel$, $Belgium$") et $hasPart(G{-}10, AulaB7)$) en appliquant à nouveau la distributivité à gauche, le résultat est une preuve de $hasLocation(AulaB7,$ "$B{-}1050, Brussel, Belgium$"). En outre, en ajoutant ces preuves aux précédantes (i.e., $hasLocation(Richard, AulaB7)$) les Σ-types suivants sont prouvés :

$$\begin{aligned} \tau_5 &= \Sigma x : room\ .\ \Sigma y : address\ .\ hasLocation(x, y) \\ \tau_6 &= \Sigma x : Person\ .\ \Sigma y : room\ .\ hasLocation(x, y) \end{aligned} \tag{2.10}$$

A la condition que la relation $hasLocation$ soit transitive et en applicant les différentes spécifications, il est facile de prouver $hasLocation(Richard,$ "$B - 1050, Brussel, Belgium$"). Ces exemples témoignent de l'expressivité des spécifications réalisables à partir des connaissances ontologiques.

3

Autres approches de représentation

La première section donne une introduction sommaire à la méthode FLUX. Méthode qui repose sur le calcul des FLUENTS (extension du calcul des situations). Nous présenterons ce langage, car l'exemple d'implantation de DTF qui sera présenté dans la partie suivante s'inspire d'une implantation réalisée en FLUX, le monde du WUMPUS.

La seconde section introduit brièvement les logiques de description pour pouvoir comparer qualitativement l'expressivité des DL et de DTF.

3.1 La méthode FLUX (Fluent Executor)

3.1.1 Généralités

Le nom de "Calcul des Fluents" provient du nom d'une théorie fondamentale appelée les Fluents. Un fluent est la représentation d'une propriété atomique du monde qui pourrait changer au fil du temps, par exemple la localisation d'un objet, une porte fermée ou ouverte ou encore la position d'un agent. Formellement les fluents sont des termes du langage. Cette représentation des propriétés par des termes est appelée la réification.

La méthode FLUX [136] est une méthode de programmation utilisée pour la modélisation des agents intelligents. Cette méthode utilise une logique qui s'appuie sur le calcul des fluents [135] pour les raisonnements. Elle est particulièrement adaptée au cas où la connaissance sur l'environnement est incomplète. Flux permet d'encoder des états incomplets par une technique de mise-à-jour de ces états qui sont décrits par des listes de fluents avec des variables, ainsi que des états de connaissances négatives et disjonctives.

L'implantation du programme qui exécute Flux comprend trois parties :

1. Le noyau qui permet le raisonnement (encode les axiomes du Fluent Calculus).
2. La théorie du domaine spécifique à une application qui inclut les axiomes d'effets associés aux actions.
3. La stratégie qui spécifie le comportement de l'agent.

Flux correspond à la classe des algorithmes de recherche utilisant des espaces d'états dont la résolution repose sur le parcours d'un arbre dont les noeuds symbolisent les états du monde. La

racine est l'état initial et les feuilles sont des solutions de l'état but. Les arcs sont la représentation des actions, et la recherche peut selon les algorithmes être faite en arrière (à partir du but) ou en avant. Dans le cas de Flux, elle est faite en avant.

Tous ces algorithmes implémentent les mêmes cycles, à savoir :

- Choix d'un noeud de l'arbre de recherche.
- Trouver un opérateur applicable à l'état en cours permettant de se rapprocher du but.
- Construction du noeud suivant, puis ajout dans l'arbre de recherche et poursuite.

Dans le cadre de Flux les hypothèses suivantes sont vérifiées :

- L'univers est statique.
- L'agent est omniscient.
- Les effets des actions sont déterministes et connus.
- Les actions sont atomiques, l'effet est immédiat et les exécutions sont ininterruptibles.

3.1.2 Exemple

Voici pour illustrer la méthode de résolution utilisée par FLUX voici un exemple donné en logique propositionnelle. Prenons cinq opérateurs E_1, E_2, E_3, E_4 et E_5, un état initial $\{e_1\}$ et enfin un état but $\{e_5\}$. Les fluents qui seront ajoutés par les actions sont préfixés par le signe $+$ et les fluents retirés par le signe $-$.
Ainsi :

Etat initial = $\{e_1\}$
But = $\{e_5\}$
Les opérateurs =
$\{E_1 : e_1 \rightarrow +e_2$
$E_2 : e_1 \rightarrow +e_3$
$E_3 : e_1 \rightarrow +e_4 - e_1$
$E_4 : e_2\ e_3 \rightarrow +e_4 - e_1 - e_2$
$E_5 : e_3\ e_4 \rightarrow +e_5\}$

En effectuant une recherche dans l'espace de recherche 3.1 avec un algorithme de recherche en profondeur d'abord, nous partons de l'état intitial, puis entuilisant le premier opérateurs applicable E_1 nous nous retrouvons dans l'état $\{e_1, e_2\}$. Dans cette état le premier opérateur applicable reste E_1 et nous bouclons (d'où l'idée de mémoriser les états explorés). En continuant dans une recherche en profondeur le plan-solution sera $\langle E_1, E_2, E_3, E_4, E_5\rangle$.

3.2 Les logiques de descriptions

Les Logiques de Description [27, 26] (DL) sont une famille de langages utilisés pour la représentation des connaissances d'un domaine d'application. La dénomination Logique de Description vient du fait que les notions importantes du domaine sont des descriptions de concepts, i.e. des expressions conçues à partir de concepts atomiques (des prédicats unaires) et des rôles atomiques (des prédicats binaires) qui utilisent les concepts et les rôles à partir des constructeurs fournis par une DL. Les DL diffèrent de leurs prédécesseurs, comme les réseaux sémantiques et les cadres

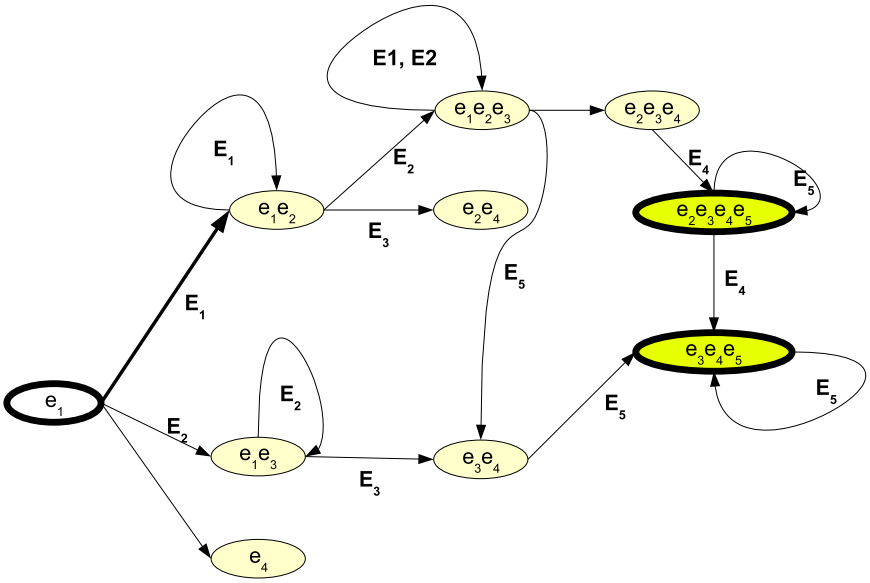


FIGURE 3.1 – Exemple d'espace de recherche du problème.

(frames), parce qu'ils utilisent une sémantique reposant sur une logique (e.g., la logique de premier ordre).

L'expression principale utilisée dans les DL est le concept de description qui décrit un ensemble d'individus ou d'objets. Formellement, un concept de description est construit à partir des noms de concepts et des noms de rôles. Le nombre des constructeurs fournis par une DL déterminera son pouvoir d'expressivité.

Les Logiques de Description sont développées pour décrire des connaissances ontologiques. Les primitives de base de la syntaxe de ces logiques sont aux nombres de trois, à savoir :

1. Les noms de concepts, e.g. médicaments, personnes.
2. Les noms de rôles, comme PrendUnMédicament.
3. La notion d'individus (les éléments d'un concept).

Pour faire un parallèle avec la théorie des ensembles, les concepts sont interprétés comme des ensembles d'éléments et les rôles comme des relations entre les éléments de ces différents concepts. Par exemple, *PrendUnMédicament* est un rôle qui permet de signaler un lien entre une personne et un médicament. Plus formellement, une interprétation sémantique de ce lien serait un paire $I = (\Delta^i, .^I)$, dont Δ est à la fois un domaine de l'interprétation et une fonction d'interprétation de $.^I$ qui fait le lien entre tout concept et un élément d'un sous-ensemble de Δ d'une part et qui fait le lien entre un rôle et un sous-ensemble de $\Delta^i \times \Delta^i$ d'autre part.

Les éléments des DL peuvent être combinés en utilisant des constructeurs pour former de nouveaux concepts et expressions (rôles). Chaque DL possède ses propres constructeurs. Cependant

toutes les DL fournissent au moins des constructeurs pour la conjonction de concepts, et la plupart fournissent également des constructeurs pour la disjonction et la négation. Les rôles peuvent être combinés aux concepts en utilisant les quantificateurs universel et existentiel. L'expression des concepts peut être manipulée dans des axiomes d'inclusion et de définition qui imposent une restriction sur les interprétations possibles de la connaissance d'un domaine.

Dans les DL, le raisonnement est principalement basé sur la subsomption entre les concepts en relations complexes.

L'expérience qui résulte de la pratique des logiques de description pour la formalisation des ontologies a révélé des problèmes de décidabilité et d'expressivité de ces langages lorsqu'ils étaient utilisés pour la formalisation de certains problèmes, tels que :

- La difficulté de définir des prédicats d'une arité quelconque (limitation aux relations unaire et binaire).
- L'impossibilité d'utiliser de la quantification sur des structures comme des arbres (la plupart des DL sont des fragments de la logique du premier ordre), ce qui peut cependant être contourné.
- La difficulté de formuler des questions autres que la subsomption et la vérification de règles de type clauses de Horn comme par exemple : $\forall x \forall y\ A(x) \wedge B(y) \ \rightarrow \ R(x, y)$.
- L'impossibilité d'exprimer des formes de connaissance non monotones.

De nombreuses recherches sont en cours pour améliorer l'expressivité des DL. Ces travaux, pour la plupart, partent du principe que le problème vient de la logique du premier ordre et regardent dans la direction des logiques non monotones et même dans la direction des logiques intuitionnistes [49].

3.3 La comparaison syntaxique entre les DL et DTF

Nous ne prétendrons pas dans cette section à une comparaison exhaustive entre les syntaxes de la DL et de DTF, mais nous envisagerons plutôt de donner un aperçu de la correspondance qui existe entre les opérateurs de base des DL et leur syntaxe en DTF.

L'intersection entre des concepts en DL (e.g., human $\sqcap$ male) se traduit par l'introduction d'une variable x de type *individual* et son insertion dans des types dépendants (voir figure 3.2). Le premier concept (human) nécessite l'introduction d'un type somme :

$$\sigma_1 \ : \ \Sigma\, x \ : \ individual\,.\,human(x)$$

Pour représenter le fait que la variable x correspond simultanement à un humain et à un mâle, il convient d'introduire ensuite un Σ-type emboité :

$$\sigma_2 \ : \ \Sigma\, x \ : \ \sigma_1\,.\,male(\pi_1 x)$$

L'union entre des concepts en DL (e.g., doctor $\sqcup$ lawyer) ce traduit encore par l'introduction d'une variable x de type *individual*. Par contre nous définissons deux Σ-types de la manière suivante :

$$\sigma_1 \ : \ \Sigma\, x \ : \ individual\,.\,doctor(x)$$
$$\sigma_1' \ : \ \Sigma\, x \ : \ individual\,.\,lawyer(x)$$

Remarquons que la variable x dans σ_1 n'a aucun rapport avec la variable x dans σ_1' hormis le fait qu'elles ont le même type. Notons également que seul un des deux sigmas sera prouvé pour un objet x donné et contrairement à la logique classique, nous pouvons dire lequel (*discernabilité des preuves*).

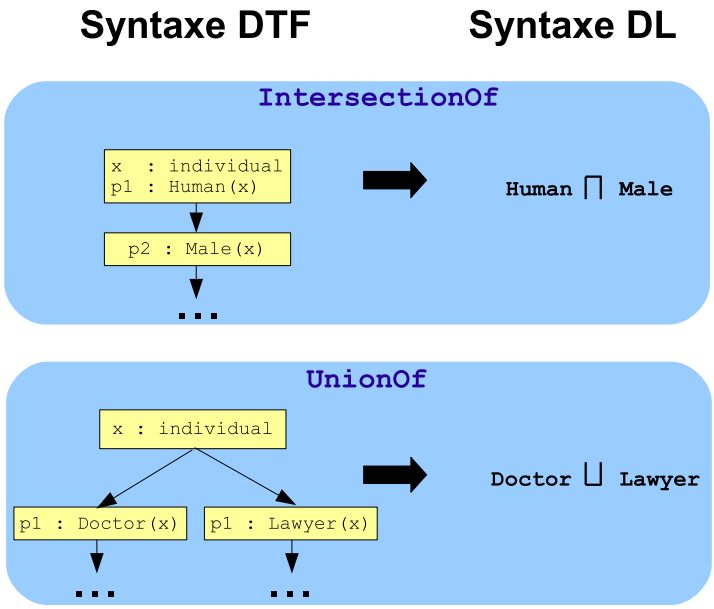


FIGURE 3.2 – Comparaison syntaxique entre DL et DTF

Le quantificateur universel en DL est usuellement représenté par une expression du genre $\forall r.c$ où r est l'expression d'un rôle où d'une propriété et c un concept. Dans le cas de la propriété *hasChild* appliquée au concept *doctor*, nous introduirons une variable x (liée) de type *doctor*, ainsi qu'une variable de type *child*, la relation entre les deux étant formalisée par le Π-type de la figure 3.3.

Le quantificateur existentiel en DL est usuellement représenté par une expression du genre $\exists r.c$ où r est l'expression d'un rôle où d'une propriété et c un concept. Dans le cas de la propriété *hasChild* appliquée au concept *lawyer*, nous introduirons une variable x de type *lawyer*, ainsi qu'une variable de type *child*, la relation entre les deux étant formalisée par le $\Sigma - type$ de la figure 3.3.

3.4 Les insuffisances des approches "classiques"

Dans une récente publication [97], l'auteur souligne l'inaptitude des méthodes classiques de représentation des connaissances à exprimer les relations de tout à parties ainsi que leurs propriétés. Il faut distinguer les problèmes de fond, qui proviennent de l'utilisation de la théorie des ensembles, des problèmes de représentation venant des langages tels que les logiques de description. Il est apparu qu'un certain nombre de ces problèmes était le résultat direct de l'emploi de la théorie des ensembles comme support de formalisation [129]. Par exemple, l'agrégation de plusieurs entités physiques est toujours une entité physique. Le problème est que la théorie des ensembles n'autorise pas la spécification de contraintes sur le type des membres des relations. En

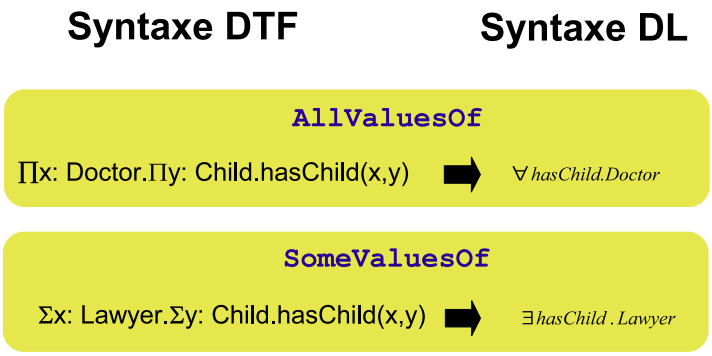


FIGURE 3.3 – Comparaison syntaxique entre DL et DTF

réalité, les ensembles représentent des abstractions, indépendantes de la nature de leurs membres, lesquels peuvent être physiques, abstraits, ou mixtes.

Il existe une différence essentielle entre un ensemble de molécules qui constituent un corps (un tout) et les molécules qui sont parties de notre corps :

1. Les éléments d'un ensemble ne possèdent aucune structure additionnelle concernant ces éléments, alors que les parties d'un tout possèdent cette caractéristique qui ne peut donc pas être représentée.
2. Lorsque les éléments d'un ensemble viennent à changer, l'extension du tout doit également être changée, bien que son identité soit maintenue.
3. Sowa et d'autres ont signalé que si l'extension d'un ensemble contenant des parties était justifiée, elle représente alors des objets physiques, et donc le tout également. Ceci est représentable par une théorie comme la méréologie, mais n'est pas garanti par l'agrégation d'ensembles.

Certaines approches sur les DL, telles que $\mathcal{ALC}$, définissent par exemple le rôle *has-part* et le formalisent comme une fermeture transitive de la relation de partonomie [123]. Malheureusement, le calcul de cette fermeture transitive se révèle comme étant *EXPTIME-complète*. Avec l'approche DL-PCC qui repose sur une logique de premier ordre, l'expressivité et la complexité sont trop limitées pour satisfaire toutes les contraintes d'un langage de type DL. S'ajoute à cela que :

i la décidabilité de leur théorie n'est pas prouvée,

ii plusieurs contraintes et définitions ne peuvent pas être représentées, telles que l'impossibilité d'établir l'irreflexivité et l'asymétrie des relations component-of, *proper-part-of* et contained-in.

Bien que la transitivité des relations partonomiques soit formalisée, aucune option ne permet d'indiquer que certaines relations partonomiques sont intransitives. Par ailleurs, dans [3] les auteurs remarquent qu'un certain nombre de difficultés sont induites par les différences de traitement entre ABox et TBox.

Enfin, dans [15], dans une analyse de la faisabilité de la représentation des relations de partonomie avec les DL, les auteurs ont conclu, que les DL n'étaient pas appropriées à la formulation

d'interrelations complexes. La raison en est que cela nécessite en réalité de disposer de deux niveaux de conceptualisation, l'un, dans lequel les relations sont décrites et le raisonnement possible avec ces relations, et l'autre, qui est un niveau supérieur (métaniveau), dans lequel une logique utilise des métadonnées qui rendent explicites les propriétés des relations du niveau inférieur. Les auteurs proposent comme solution d'utiliser une logique du premier ordre pour les deux niveaux, mais il reste alors le problème de la communication des informations entre ces niveaux. Nous proposons une solution dans laquelle les deux niveaux de conceptualisation sont intégrés dans un même langage.

4

Conclusion de la partie 2

Avec le formalisme de McCarthy, les situations sont interprétées comme des prédicats, et pour les prédicats dépendants d'une situation, la situation apparaît comme un nouveau paramètre des prédicats. Les règles d'extension (lifting) concernent les relations entre les prédicats absolus qui sont indépendants de la situation. Au contraire, dans DTF, les axiomes d'extension ne sont pas utiles. Ce sont les Contextes qui sont des citoyens du premier ordre et non pas les situations. Par conséquent, les situations sont des preuves et la description des connaissances est une représentation indépendante de la situation.

D'autres formalismes, tels que les logiques de description, ne permettent pas, par manque d'expressivité, de représenter tous les concepts du domaine. En particulier, il est impossible de définir les propriétés complexes résultant de la composition d'autres propriétés. Cette limitation provient de l'absence de variables typées et de l'impossibilité de définir des jointures [75]. En revanche DTF permet, par la relation de tout-à-partie définie par des Σ-types, de représenter la composition des fonctions.

Un ingrédient essentiel pour représenter la dynamique des situations est une forme de partialité, ce que les DL ne peuvent pas exprimer, puisqu'elle repose sur la logique classique. A contrario, la logique constructive (fondation de DTF) possède, de manière intrinsèque, cette notion de partialité (validité partielle).

Dans les domaines de l'anatomie et du médical, les dépendances entre les propriétés ontologiques et entre les ontologies et les prédicats d'autres domaines sont nécessaires. Dans [67] les auteurs expliquent que ni OWL, ni SWRL ne sont capables de capturer les subtilités de ces dépendances. Avec les types dépendants, il est possible de définir des modèles plus précis et plus concis. DTF peut être vue comme une théorie basée sur une logique de dépendances entre les termes avec laquelle des théories modulaires et inter-dépendantes peuvent être interconnectées.

Parce qu'elle est une logique d'ordre supérieur, DTF autorise la définition de propriétés sur les types eux-mêmes à la manière des descriptions méta (cf. 4.1). Cela permet de réaliser des dérivations et donc des raisonnements sur les types, i.e. prouver par exemple qu'une relation est transitive et en déduire d'autres propriétés. Avec DTF, les problèmes classiques posés lors de la conception d'applications à contextes ouverts peuvent être résolus par des solutions précises dans un cadriciel cohérent. Ainsi, l'approche de la représentation par une théorie des types est plus naturelle qu'avec une théorie logique du premier ordre.

TABLE 4.1 – Ce tableau compare les principales formalisations des contextes (ST : la théorie des situations, SC : le calcul des situations, FOL : la logique du premier ordre, CG : les graphes conceptuels, IM : modélisation de l'information).

	ST	SC	FOL	CG	IM	**DTF**
CFCO	+	+	+	-	+	+
CSV	+	-	+	-	+	+
CSS	+	+	+	+	+	+
Nest	+	-	-	+	+	+
DMO	-	-	+	+	+	-
Inh	+	-	-	-	+	+
SR	+	-	-	-	+	+
SPC	-	-	-	-	-	+

CFCO : les contextes sont des objets de premier classe.
CSV : vocabulaire spécifique au contexte.
CSS : sémantique spécifique au contexte.
Nest : emboîtement des contextes.
DMO : Difficulté de mise en oeuvre.
Inh : héritage entre les contextes.
SR : auto-référence.
SPC : spécification des propriétés des contextes.

Troisième partie

Mise en oeuvre expérimentale de DTF

1

Le monde du Wumpus

1.1 Présentation du problème

Il existe une grande variété d'applications qui sont appelées "mondes" et utilisées comme exemples pour la représentation des connaissances, des raisonnements et du planning en intelligence artificielle. Parmi elles, les plus connues sont le monde des blocs (bloc world), le monde de l'aspirateur et le monde du Wumpus. Après avoir étudié une implantation du monde du Wumpus récemment réalisée en Flux, nous avons décidé d'expérimenter DTF avec cette application.

Michael Genesereth introduisit le Monde du Wumpus comme un jeu au début des années 80. Puis Russel et Norvig en proposèrent une application comme illustration du fonctionnement des agents autonomes dans leur livre [121]. Le principe de l'application est de déplacer un agent autonome, dans un espace dynamique discret (appelé cave) et accessible, d'une manière non déterministe, pour la recherche d'un sac d'or.

Ce monde est représenté par des grilles (e.g. figure 1.1) dont la taille peut varier. Ces grilles peuvent contenir des trous dans lesquels l'agent peut tomber et ainsi mourir. Les trous sont indiqués par des souffles présents dans des cases adjacentes à celles où ils se trouvent. De même, certaines cases peuvent contenir un animal appelé "Wumpus" susceptible de dévorer l'agent. Le Wumpus ne peut pas se déplacer, mais il peut être tué par l'agent à l'aide d'une flèche. Les cases adjacentes à celles du Wumpus dégagent une odeur particulière. Une seule case du jeu contient de l'or. Elle peut éventuellement être une case sur laquelle se trouve un Wumpus, mais pas un trou. L'objectif de l'agent sera bien évidemment de trouver l'or.

Dans le cas étudié dans ce mémoire, le jeu s'arrêtera là, alors que, dans le jeu original, l'agent devait sortir de la grille par un retour à la case départ. Le principe du jeu est essentiellement de réussir cette opération en un minimum de déplacements et un score le plus élevé possible.

Notre but n'est pas de proposer une solution plus performante que les algorithmes de recherche actuels, mais de montrer simplement qu'il est possible, en utilisant un automate écrit en DTF, de modéliser clairement et simplement le problème pour le résoudre d'une manière à la fois élégante et expressive.

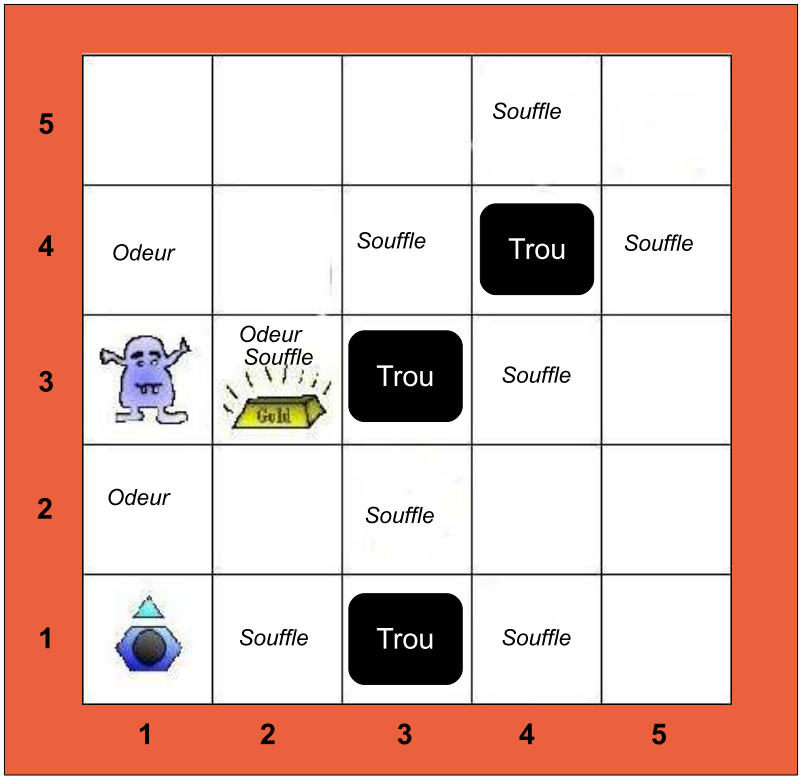


FIGURE 1.1 – Exemple d'une grille 5x5 du monde du Wumpus.

1.2 Principe de la solution en Flux

Voici l'exemple d'une implantation du monde du Wumpus dans le langage FLUX, qui permet de faire une comparaison entre la définition formelle du problème exprimée par une logique du premier ordre et le même problème résolu par DTF. Ceci, afin de faire ressortir le manque d'expressivité et d'évolutivité des représentations par les logiques du premier ordre.

Une version de base du monde du Wumpus peut être axiomatisée par neuf fluents : Ce sont At(x,y) et Facing(d) qui représentent respectivement le fait qu'un agent est dans la case en position (x,y) et qu'il se dirige dans la direction d (correspondant aux quatre points cardinaux). Les axiomes Gold(x,y), Pit(x,y) et Wumpus(x,y) sont des axiomes de position, ils permettent d'indiquer les positions respectives de l'or, des trous et du Wumpus. L'axiome Dead indique que le Wumpus a été tué. Enfin, les axiomes Has(x), XDim(n) et YDim(n) représentent le fait qu'un agent possède de l'or ou une flèche, $x \in \{gold, arrow\}$ et la taille, initialement inconnue de la grille.

Flux permet entre autres possibilités de combiner la spécification des effets physiques et ceux obtenus par les capteurs en un seul prédicat. Par exemple, stateUpdate(z_1,A(x),z_2,y) décrit l'état mis à jour de z_1 et z_2 dépendant de l'effet physique d'une action A(x) et du résultat donné par le capteur Y.

Après l'initialisation du système et l'exécution de la première action "tourne à gauche", l'exploration de la grille par l'agent commence. Le programme utilise trois paramètres entrants, à savoir, les points explorés, les cases explorées et enfin le chemin actuel, ce dernier permettant de revenir en arrière. Le choix des directions dépend d'une liste qui encode les quatre directions possibles. L'agent sélectionne le premier élément de la liste en cours. Si la direction peut être explorée et que l'agent n'est pas resté à la même place (ce qui indiquerait que l'agent est entré dans un mur), il vérifie tout d'abord s'il a trouvé de l'or. Si c'est le cas, il recherche le chemin le plus court pour revenir à son point de départ. Sinon, il vérifie que le Wumpus peut être éliminé. Alors un nouveau point est ajouté dans l'arbre indiquant la position actuelle. L'agent teste une nouvelle direction. Si la liste courante des directions est vide, l'agent rebrousse chemin. Le prédicat Backtrack permet à l'agent de reculer d'un pas sur le chemin courant en inversant sa direction, et le programme se terminera lorsque le chemin sera vide, signifiant que l'agent est revenu à sa position de départ et qu'il a visité et nettoyé le plus grand nombre de cases possible.

Si le prédicat auxiliaire $Explore(x, y, d, v, z_1, z_2)$ est vérifié, l'agent peut sans problème aller depuis sa position actuelle (x,y) dans une direction d. Une direction pourrait être explorée dans le cas où la case adjacente n'est pas un élément de la liste v des cases déjà explorées et n'est pas indiquée comme étant hors des limites. De plus, la case adjacente ne doit pas être reconnue comme étant un trou ou contenant le Wumpus.

Voici un exemple de représentations :

Détection d'une case avec une odeur indiquant la présence du Wumpus dans une des cases adjacente

stenchPerception(X,Y,Percept,Z) : −
$XE\# = X + 1, XW\# = X - 1, YN\# = Y + 1, YS\# = Y - 1,$
$(Percept = false- >$
$not\_holds(wumpus(XE, Y), Z),$
$not\_holds(wumpus(XW, Y), Z),$
$not\_holds(wumpus(X, YN), Z),$
$not\_holds(wumpus(X, YS), Z)$
$; Percept = true,$
$or\_holds([wumpus(XE, Y), wumpus(X, YN),$
$wumpus(XW, Y), wumpus(X, YS)]$
$, Z)).$

Détection d'une case avec un souffle indiquant la présence d'un trou dans une des cases adjacente

breezePerception(X,Y,Percept,Z) : −
$XE\# = X + 1, XW\# = X - 1, YN\# = Y + 1, YS\# = Y - 1,$
$(Percept = false- >$
$not\_holds(pit(XE, Y), Z),$
$not\_holds(pit(XW, Y), Z),$
$not\_holds(pit(X, YN), Z),$
$not\_holds(pit(X, YS), Z)$
$; Percept = true,$
$or\_holds([pit(XE, Y), pit(X, YN),$
$pit(XW, Y), pit(X, YS)], Z)).$

Détection d'une case avec un éclat indiquant la présence d'or **glitterPerception**(X,Y,Percept,Z)
$:-$
$Percept = false -> not\_holds(gold(X,Y),Z)$
$; Percept = true, holds(gold(X,Y),Z).$

Pour des explications détaillées de ces exemples l'on pourra consulter [134].

L'agent ne chasse les Wumpus que dans des endroits connus, et, pour des raisons de simplification, seulement lorsqu'il est dans la même ligne ou la même colonne que ceux-ci. Lorsque le Wumpus est mort, l'agent peut explorer des zones qu'il avait ignorées précédemment. Pour cela, les cases qui se trouvent dans la liste v des cases déjà explorées entre l'agent et le Wumpus mort peuvent être revisitées.

Après avoir trouvé l'or, l'agent utilise le prédicat GoHome pour se diriger vers la sortie par le chemin le plus court. Pour l'aider dans cette tâche, il dispose du prédicat StarPlan qui lui permet d'utiliser un algorithme de type A^* et la distance de Manhattan comme fonction heuristique.

Dans les chapitres suivants nous réaliserons une implémentation complète du Wumpus en DTF, ce qui permettra une comparaison avec la méthode Flux.

2

Réalisation d'un simulateur DTF

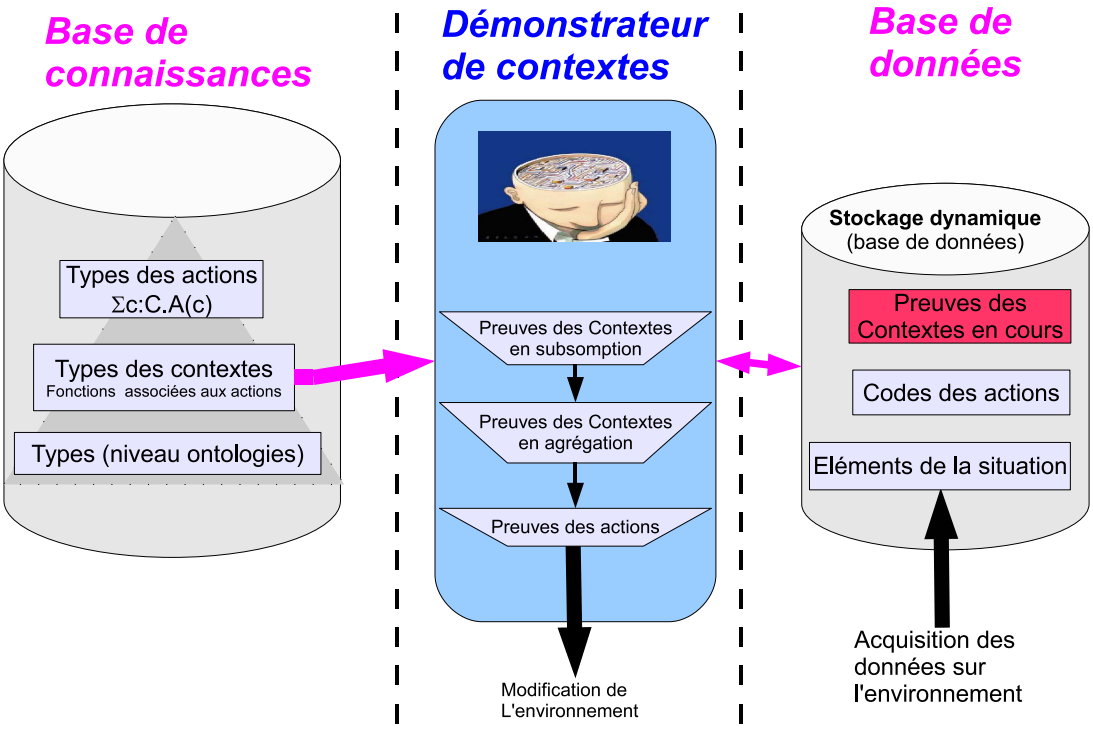


FIGURE 2.1 – Démonstrateur de DTF

L'agent DTF est un programme divisé en deux parties (représentées à la figure 2.1) : la première est la représentation ontologique de la connaissance (la définition des types de DTF) et la seconde l'algorithme de démonstration (recherche de preuves des Contextes et des Actions) et la base de données (la situation).

2.1 Aspect ontologique de la connaissance

La partie ontologique répond aux différentes définitions des types de base et des types dépendants (Contextes et Actions). Elle correspond au modèle purement dynamique de la représentation du monde accessible au programme. Elle peut évoluer en cours d'exécution.

Les types de base sont les concepts d'une ontologie de domaine. Les relations ontologiques entre les concepts sont représentées par des types dépendants (Σ et Π types). Par exemple, les types personne et ingénieur s'écrivent respectivement $Personne : Type_1$ et Ingénieur: $Type_1$. Le lien de subsomption entre les deux, à savoir que *Ingénieur* est un concept subsumé par *Personne* peut s'exprimer en utilisant le constructeur $Sub : Sub(Personne, Ingénieur)$.

Les contextes sont construits comme des Σ-types et des Σ-types emboîtés à partir des types de base et des relations ontologiques définies dans la couche des types. Les prédicats d'actions sont les Π-types, signatures des fonctions utilisées pour réaliser les actions.

Un exemple de la définition d'une fonction d'action est :

keep-gold $\triangleq (\Pi\, a : agent\,.\,(\Pi\, b\ :\ gold\,.\,(keep(a,b))))$ où keep est une fonction dépendante du type $agent \rightarrow gold \rightarrow Prop$.

Enfin, les définitions des types d'actions sont des Σ-types définis à partir des Contextes et des prédicats d'actions de la couche précédente.

2.2 Représentation de la situation et algorithme de démonstration

La seconde partie correspond à l'algorithme de démonstration proprement dit et à la situation. Les éléments de la situation constituent les preuves des types définis dans la couche de base de la partie ontologique. Elle peut être mise à jour par exemple en utilisant des capteurs symboliques.

Le code d'une action est un code local qui peut être écrit dans un langage différent de celui utilisé pour le démonstrateur. Il peut par exemple correspondre à un web service ou un code de bas niveau spécifique à des actionneurs particuliers.

La base de la dérivation des preuves des contextes en cours est un résultat temporaire enregistré pendant un cycle du démonstrateur. Elle est enrichie par les algorithmes de preuves des Contextes et des Contextes agrégés. Puis, elle est utilisée par les algorithmes de preuves des Contextes agrégés et des preuves des actions.

2.3 Principes du démonstrateur

Voici une implantation possible du démonstrateur de DTF (appelé DTF-A). L'objectif de cette implantation est de montrer qu'elle est réalisable par un algorithme avec une complexité

raisonnable.

Le principe du fonctionnement du démonstrateur DTF-A pour les contextes se présente de la manière suivante :

Définition 37. (jointure) La jointure est une opération qui porte sur deux relations R1 et R2, et à partir de celles-ci, construit une troisième relation qui regroupe exclusivement toutes les possibilités de combinaison des occurrences des relations R1 et R2 satisfaisant l'expression logique EL. La jointure se note R1 $\rhd\lhd_{EL}$ R2.

Définition 38. ($\theta-jointure$) Une $\theta-jointure$ est une jointure pour laquelle l'expression logique EL est une simple comparaison entre un attribut A1 de la relation R1 et un attribut A2 de la relation R2. La $\theta-jointure$ est notée R1 $\rhd\lhd_{EL}$ R2.

Par exemple, supposons que le constituant *sol* soit commun aux relations *plantes* et *régions*, alors la θ-jointure de *plantes* et *régions* par rapport à l'égalité de *sol* dans chacune aura pour résultat toutes les plantes compatibles avec le sol d'une région : plantesRégionnal = plantes $\rhd\lhd_{sol_{plantes}\theta sol_{rgions}}$ régions.

Définition 39. (équi-jointure) Une équi-jointure est une $\theta-jointure$ dans laquelle l'expression logique EL est un test d'égalité entre un attribut A1 de la relation R1 et un attribut A2 de la relation R2. L'équi-jointure est notée R1 $\rhd\lhd_{A1,A2}$ R2.

Soit en entrée : un domaine fini S (la situation), un ensemble de contextes types C et un ensemble fini de Types T = $T_1, T_2, ..., T_i$ et de propositions P = $p_1, p_2, ..., p_n$ sur T.

Le démonstrateur repose sur quatre algorithmes : le premier et le deuxième sont utilisés pour démontrer qu'un contexe et un contexte agrégé sont prouvés dans la situation en cours. Le troisième permet de rechercher dans l'arbre des preuves des contextes de la situation en cours. Enfin le quatrième est le démonstrateur DTF-A proprement dit.

Voici le détail de ces algorithmes :

1. Le premier algorithme permet de vérifier si un type Contexte peut être prouvé dans la situation courante. Pour cela, l'algorithme parcourt un arbre de résolution. La démonstration d'un type Contexte revient à évaluer les sommets de l'arbre. Ces évaluations sont, soit des $\Sigma-types$, c'est-à-dire des associations déjà représentées en mémoire par des tables d'association, soit des $\Pi-types$, c'est-à-dire des fonctions qui correspondent à des restrictions faites sur les occurrences d'un Type. Un exemple d'un graphe de résolution est présenté à la figure 2.2.

   Il est la représentation du Contexte donné par le Σ-type suivant :

   ProblèmeTV $\triangleq \Sigma\, x : lecteurDvd\,.\,\Sigma\, y : TV\,.\,\Sigma\, z : image\,.\,\Sigma\, p_1 : neuf(x)\,.\,\Sigma\, p_2 : connexion(x,y)\,.\,\Sigma\, p_3 : absenteDe(z,y)$

Principe de la résolution, le noeud (i) correspond à la collection de preuves des associations entre un poste de TV et un lecteur de DVD. Le noeud (ii) correspond à la collection de preuves des TV qui ne possèdent pas d'image. Le noeud (iii) sélectionne les éléments de la collection (i) qui sont neufs. Enfin le noeud (iv) est une équi-jointure entre l'élément TV des deux collections de preuves de (i) et de (iii). Cette équi-jointure est représentée à la figure 2.2.

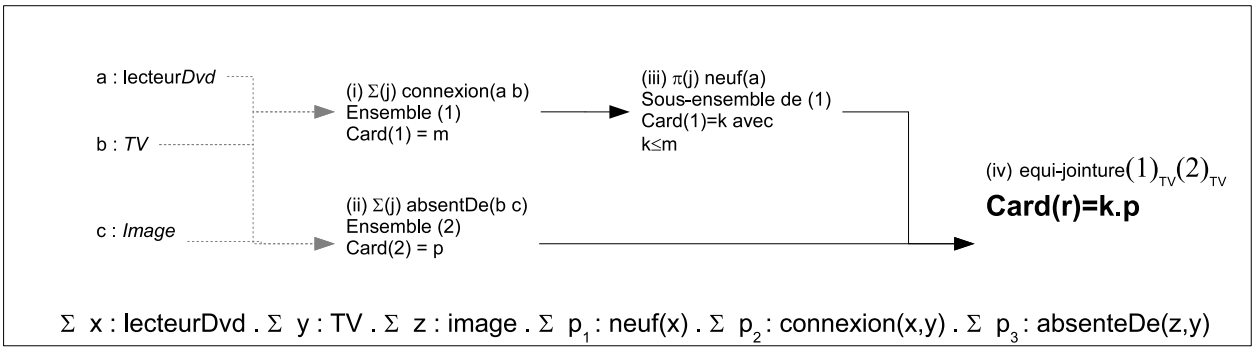


FIGURE 2.2 – Arbre de résolution (algorithme 1)

2. Le deuxième algorithme recherche, à partir des preuves des Contextes à prouver les Contextes agrégés (CA). Pour cela, il explore à la manière de l'algorithme 1 les arguments qui sont ici les preuves des Contextes (i.e. la liste des types des CA enfants des contextes validés). Lorsqu'un CA est prouvé, il est ajouté à la liste des CA.

3. Le troisième algorithme parcourt la liste des partonomies d'un contexte valide donné et vérifie, avec l'algorithme 1, s'ils peuvent être prouvés (i.e. recherche des objets de ce type). Si un contexte a pu être prouvé par une ou plusieurs instances, alors l'algorithme continue d'explorer les partonomies de celui-ci par un appel récursif.

4. Le quatrième algorithme désigne l'automate proprement dit. Il correspond à la boucle d'exécution du programme qui procède en cinq étapes :
   - i Exécution de l'algorithme 1 à la recherche des Contextes démontrés.
   - ii Exécution de l'algorithme 2 à la recherche d'éventuelles preuves de Contextes agrégés.
   - iii Inférence de nouveaux concepts par un raisonnement sur la situation et les Contextes vérifiés.
   - iv Exécution des actions associées aux contextes prouvés.
   - v Suppression des preuves des contextes en cours (sous-types du contexte racine).

   Une fois qu'il a effectué les cinq étapes, l'algorithme recommence et continue sa recherche de preuves tant que le but actuellement défini n'est pas atteint.

input : Nom du contexte à prouver
1 Rechercher les arguments du Contexte.
2 Rechercher les preuves des arguments du Contexte.
3 **if** *il existe une preuve pour chaque argument* **then**
4 | *Le calcul présenté ici correspond à l'implantation en Lisp. qui fut réalisée par un produit cartésien et non par le parcours. d'un arbre comme indiqué précédemment par manque de temps.* Calculer la liste des arguments possibles (produit cartésien des arguments).
5 | Vérifier la validité du Contexte pour chaque élément de la liste.
6 | Ajouter les Contextes prouvés dans la liste des preuves.
7 **end**
8 return La liste des preuves des Contextes

Algorithme 1 : Preuves des Contextes

1 Rechercher la liste des Contextes agrégés à partir de la liste des Contextes.
2 Vérifier d'après le résultat les Contextes agrégés prouvés.
3 Etablir la liste des Contextes agrégés de niveau le plus bas dans la hiérarchie partonomique.
4 return La liste des Contextes agrégés prouvés

Algorithme 2 : Preuves des contextes agrégés

input : context-root level
1 Rechercher la liste des éléments enfants du Contexte courant. Pour chaque élément de la liste, vérifier s'il est prouvé sinon le supprimer.
2 **if** ¬ *(La liste des contextes n'est pas vide)* **then**
3 | **for** *i=1* **to** *taille de la liste* **do** Recherche la preuve des enfants du Contexte i
4 **end**

Algorithme 3 : Recherche des Contextes prouvés

1 **while** *L'agent est actif* **do**
2 | Rechercher les preuves des Contextes.
3 | Si le but est vérifié alors arrêter l'agent.
4 | Rechercher les preuves des Contextes agrégés.
5 | Exécuter les actions.
6 | Inférer de la connaissance.
7 | Supprimer les Contextes prouvés (et agrégés).
8 **end**

Algorithme 4 : Le démonstrateur de DTF

3

Description du prototype de l'agent

Le principe du démonstrateur consiste essentiellement en un parcours de l'arbre des contextes. L'avantage de cette solution tient au fait que les contextes parcourus pendant le fonctionnement de l'agent sont circonscrits par l'expression des contextes du domaine, ce qui bien évidemment diminue grandement l'espace de recherche. Le problème est donc de commencer à créer l'arbre des contextes. Pour cela, il n'existe pas de solution mécanique pour déduire cet arbre à partir des Contextes, comme il serait possible de le faire dans le cas d'un modèle relationnel par exemple. Le principe de la conception tendrait plutôt à s'orienter vers une correspondance entre l'expression du problème par un énoncé et les contextes. Cet aspect était à la base de la conception de DTF et des travaux futurs doivent être menés dans cette direction. En effet, les principes de DTF sont partiellement inspirés par les travaux des linguistes sur les DRT [35].

Pour vérifier le prototype de l'agent DTF, nous avons réalisé un programme qui implante les algorithmes (sauf le parcours des arbres qui est un simple produit cartésien) et nous l'avons testé avec un exemple simple : le monde du Wumpus. Les contextes sont modélisés en partant de la définition textuelle suivante :

Un agent est localisé dans une case et il est orienté dans une direction. L'agent est soit en possession de l'or et dans ce cas il doit rentrer, soit il n'est pas en possession de l'or et l'agent peut être dans une ou plusieurs des positions suivantes :

1. L'agent est sur une case qui dégage une odeur de Wumpus.
2. L'agent est sur une case d'où il peut sentir le souffle d'un trou.
3. L'agent se trouve dans une case dans laquelle se trouve de l'or.
4. L'agent est dans une case dont il sait que la position suivante est sans danger (par déduction préalable et enrichissement de ses connaissances).
5. L'agent n'est dans aucune de ces configurations.

Dans le cas 1, l'agent changera de direction. Dans le cas 2, l'agent pourra, en vérifiant plus précisément dans la base de connaissances, prouver qu'il connaît la case sur laquelle se trouve le Wumpus. Il pourra alors, si le Wumpus n'est pas mort, le tuer. Sinon il peut accéder sans danger à la case suivante. S'il n'a pas pu prouver la présence du Wumpus, l'agent devra changer de direction. Dans le cas 3, l'agent s'emparera de l'or. Dans le cas 4, l'agent avancera d'une case. Dans le cas 5, l'agent changera de direction.

Maintenant, un agent peut éventuellement se trouver dans une situation où plusieurs preuves sont validées en même temps. Par exemple, il est dans une case qui vérifie les cas 1, 2 et 4. Pour simplifier, lorsque le cas se présente, l'algorithme peut choisir un Contexte agrégé. Le cas donné en exemple correspond donc avec un Contexte agrégé qui, associé avec un prédicat d'action (ici le prédicat avancé d'une case dans la direction en cours), permettra de simplifier le schéma de conception.

La conception des types du Wumpus passe par la définition des prédicats d'actions, puis à partir de ces prédicats, nous identifions les contextes qui leurs sont associés, le lien entre les actions et les contextes constitue le type action. Le résultat formel (i.e., la liste des Contexes et des actions obtenues à partir de l'énoncé textuel du paragraphe précédent) est détaillé à la figure 3.1.

3.1 L'algorithme

L'algorithme du démonsrateur de théorème est une fermeture dont le code est entièrement fourni en annexe.

```
(defun proof-automath()
    (let ((i 0))
    (loop while (equal active-agent t)
          do(progn
              (setf i (+ i 1))
              (get-proofs-of-the-current-situation current-type 1)
               (if (is-goal-valid)
                    (progn
                      (print "Ok goal !")
                      (set-stop-active-agent))
                    (progn
                      (set-lst-of-the-deepest-dtr)
                      (print "lst-of-deepest-context")
                      (print lst-of-deepest-context)
                      (setf  lst-of-compound (get-proofs-of-current-compound-dtr))
                      ;;(print "lst-of-compound")
                      ;;(print lst-of-compound)
                      (execute-actions)
                      (execute-deductions)
                      (erase-proof)
                      ;;(if (= 10  i) (set-stop-active-agent))
                      (print i)
                      (print "Direction : ")
                      (print (direction current-agent))
                      (print "Position X")
                      (print (coordinateX (positionG current-agent)))
                      (print "Position Y")
                      (print (coordinateY (positionG current-agent)))))
               )
      ))
    )
```

TABLE 3.1 – Algorithme LISP du démonstrateur

3.2 Choix d'un langage de programmation

Le choix d'un langage de programmation fut la première étape de la réalisation du démonstrateur de Contextes. Le paradigme fonctionnel s'imposa immédiatement grâce à la logique sous-jacente qui repose sur le $\lambda - calcul$. Mais, d'autres contraintes devaient être prises en compte avant de prendre une décision. Ce sont :

1. la portabilité,
2. le typage dynamique,

3. la liaison dynamique,
4. les primitives d'entrées/sorties,
5. les fermetures,
6. les λ expressions.

Au début, les recherches s'orientèrent vers un langage très utilisé en France dans le domaine de la recherche, le langage OCAML et ses dérivés appliqués à la démonstration de théorème comme LEGO [94] et COQ [36]. Dans le cas des démonstrateurs de théorèmes, il s'avéra que les programmes produits étaient peu performants et surtout peu portables. Il faut ajouter à cela que ce sont des applications interactives. Dans le cas de OCAML, ce langage ne permettait pas de produire du code dynamique sans recourir à des chaînes de caractères et cela obligeait le concepteur à avoir recours au pattern matching. Il ne possèdait pas non plus de typage dynamique. Le choix s'est donc logiquement porté sur un langage de la famille Lisp.

Deux implantations furent réalisées du démonstrateur de DTF, les deux codées avec le langage Lisp.

La première solution repose sur une représentation des connaissances réalisée en RDF où les interrogations de la base sont faites par des requêtes SPARQL [114]. Les descriptions des $\Sigma - types$ sont alors en correspondance avec une requête et les relations de sous-typages sont des relations RDF.

La deuxième solution est entièrement réalisée en Lisp et utilise des tables d'association pour la représentation de la base de connaissances.

L'implantation de la base de données (figure 3.3) est réalisée en utilisant des tables d'association dont les clefs sont les labels des types et les enregistrements sont des listes de preuves. Ces labels sont des expressions LISP dont la déclaration repose sur une fermeture.

3.3 Implantation en Lisp

3.3.1 Implantation des Contextes et des Contextes agrégés

Il est utile de rappeler tout d'abord que le but du démonstrateur est essentiellement de trouver des λ termes, preuves d'un type donné (type inhabitation problem). Il est donc indispensable de trouver une règle de correspondance entre d'une part l'expression de la structure et d'autre part une λ expression en LISP.

Le "let*" [23] est utilisé pour définir les variables liées $\Sigma - types$. Cela permet d'exprimer des instances de types dépendants. Pour le calcul de la preuve (par la β-réduction) d'un $\Sigma - type$ une règle est ajoutée à la conception des types. Ceux-ci doivent définir les types non dépendants en premier. Ils forment donc les arguments de la λ expression. Pour mettre en lumière la correspondance entre un type et une λ expression, voici un exemple :

Soit le type Σa : $Person\,.\,direction(a)$ par la correspondance expliquée précédemment, celui-ci devient la λ expression Lisp :

23. Le *let* permet de créer une liaison entre plusieurs fonctions (fermeture) ou à l'intérieur d'une fonction. La variable est alors locale à une ou des fonctions. La fonction *let** est semblable à la fonction let à la différence qu'elle permet de faire des initialisations de variables dépendantes des précédentes.

```
(1) (lambda x
(2)    (if (typep x 'person)
(3)        (let*
(4)            ((a x)
(5)            (b (direction a)))
(6)            (if (and (not (or (equal a nil) (equal b nil)) (typep b 'direction))
(7)            (list a b))
```

Ce code s'interprète de la manière suivante : le λ terme prend comme argument x qui est du type person, x est affecté à la variable a qui permet d'affecter une valeur du type direction à b (par le prédicat $direction(a)$), puis, si a et b ont pu être prouvés, alors la fontion retourne une liste contenant a et b, sinon elle retournera $nulle$.

Cette λ expression est stockée dans une fermeture (closure) et porte le nom de "règle d'élimination".

Les $\Sigma - types$ sont définis en LISP par une fermeture [68]. Une petite explication préalable est nécessaire pour dire pourquoi utiliser des fermetures. La raison en est simple. Si pour la représentation des concepts de l'ontologie de domaine, les objets sont utilisés, car ils ne sont qu'une représentation temporaire, il n'en est pas de même de la réalisation du démonstrateur qui, elle, est définitive et doit pouvoir être portée sur des machines comme des téléphones portables ou des systèmes embarqués. Il est alors possible de se passer des packages CLOS [2] de LISP qui demande beaucoup de ressources mémoire et processeur. Un exemple de la structure adoptée pour la représentation en LISP des $\Sigma - types$ est montré à la figure 3.3.1

3.3.2 Implantation des actions

La représentation des actions est réalisée de la même façon que pour les contextes. La structure est identique et utilise la règle d'élimination. La seule particularité, pour améliorer les traitements est l'utilisation d'une table d'association qui représente les liens entre les Contextes ou les Contextes agrégés et les Actions. Ceci pour éviter de parcourir toutes les actions, lorsque le contexte prouvé est connu.

3.3.3 Les concepts

Pour les besoins du prototype l'ontologie de domaine des concepts du Wumpus a été implantée en représentant les concepts et leurs propriétés par des objets (ce sont des $\Pi - types$) et la relation de subsomption (sous-type entre les classes). Les rôles (des $\Sigma - types$) sont représentés par des tables d'association.

3.3.4 Liste des contextes

Les contextes sont au nombre de 10 (avec le contexte racine). En voici la liste avec le type de chaque contexte :

1. **agent-location** $\triangleq$
   $\Sigma\, a : agent - Wumpus\,.\,\Sigma\, b : agent - position(a)\,.\,agent - direction(a)$

```
;;;; Created on 2008-10-09 15:15:04
;;;;  breeze-square
;;;;(load "/home/patrick/Thesis/IntContextType/v02a/IntContextType002
/IntContextType002/Ontologies/Contexts/DTR/tdr-breeze-square.lisp")

(let ((name 'breeze-square)
        (lst-types '(~agent-has-gold))
        (partOf '(~agent-has-gold))
        (hasPart '())
        (meronym '(pr1 pr2 pr3 pr4))
        (holonym '())
        (current-value '())
        (level 3)
        (constructor '(lambda (x)
                        (let*((a x)
                              (b (is-breezy-position-p (pi-1(pi-2(pi-1 a)))))
                              )
                              (if (and (not(equal b nil))
                                       (list a b)
                                       nil
                              )))))
  (defun get-level-breeze-square()
    level
    )
  (defun initialise-breeze-square()
    (setf current-value '())
    )
  (defun get-part-breeze-square ()
    partOf
    )
  (defun get-has-part-breeze-square ()
    hasPart
    )
  (defun get-meronym-breeze-square ()
    meronym
    )
  (defun get-holonym-breeze-square ()
    holonym
    )
  (defun proof-breeze-square ()
    ;;Call a generic function.
    )
  (defun get-type-lst-breeze-square ()
    lst-types
    )
  (defun breeze-square (agent-location)
    (funcall (eval constructor) agent-location)))
```

TABLE 3.2 – Exemple de codage en LISP d'un DTF

2. **agent-has-gold** $\triangleq$
   $\Sigma\, a : agent - location\,.\, has - gold - p(\pi_1 a)$
3. $\sim$ **agent-has-gold** $\triangleq$
   $\Sigma\, a : agent - location\,.\, \sim has - gold - p(\pi_1 a)$
4. **gold-square** $\triangleq$
   $\Sigma\, a :\sim agent - has - gold\,.\, is - perceive - glitter - p(\pi_1(\pi_2(\pi_1 a)))$
5. **breezy-square** $\triangleq$
   $\Sigma\, a :\sim agent - has - gold\,.\, is - breezy - position - p(\pi_1(\pi_2(\pi_1 a)))$
6. **smelly-square** $\triangleq$
   $\Sigma\, a :\sim agent - has - gold\,.\, is - smelly - position - p(\pi_1(\pi_2(\pi_1 a)))$
7. **position-in-valid-square** $\triangleq$
   $\Sigma\, a :\sim agent - has - gold\,.\, is - valid - position - p(\pi_1(\pi_2(\pi_1 a))\pi_2(\pi_2(\pi_1 a)))$
8. **next-position-is-a-valid-square** $\triangleq$
   $\Sigma\, a :\sim agent - has - gold\,.\, is - possible - direction(\pi_1(\pi_2(\pi_1 a))\pi_2(\pi_2(\pi_1 a)))$
9. **identify-Wumpus-square** $\triangleq$
   $\Sigma\, a : smelly - square\,.\, \Sigma\, b : position\,.\, \Sigma\, c : position\,.\, \Sigma\, d : smelly\,.$
   $\Sigma\, e : is{-}adjacent{-}position{-}p(\pi_1(\pi_2(\pi_1 a))b)\,.\, \Sigma\, f : next{-}position(\pi_1(\pi_2 a)(\pi_1(\pi_2(\pi_2 a))b)\,.\, \Sigma\, g :$

$is-adjacent-position-p(bc)\,.\,\Sigma\,h\,:\,is-smelly-position-p(bd)\,.\,\Sigma i\,:\,is-different-position-p(\pi_1(\pi_2 a)c)$

10. **identify-dead-Wumpus-square** $\triangleq$
    $\Sigma\,a\,:\,identify-Wumpus-position\,.\,\Sigma\,b\,:\,Wumpus\,.$
    $\Sigma\,c\,:\,is-located-in-p(\pi_2(\pi_1(\pi_1 a))b)\,.\,is-dead-p(b)$

Les contextes agrégés, eux, sont au nombre de 6, et leurs types sont :

1. pr1 $\triangleq$
   $\Sigma\,a\,:\,next-position-is-a-valid-square\,.\,\Sigma\,b\,:\,breeze-square\,.\,part(a,b)$
2. **pr2** $\triangleq$
   $\Sigma\,a\,:\,breezy-square\,.\,\Sigma\,b\,:\,gold-square\,.\,part(a,b)$
3. **pr3** $\triangleq$
   $\Sigma\,a\,:\,next-position-is-a-valid-square\,.\,\Sigma\,b\,:\,breeze-square\,.\,\Sigma\,c\,:\,smelly-square\,.\,part(a,b,c)$
4. **pr4** $\triangleq$
   $\Sigma\,a\,:\,smelly-square\,.\,\Sigma\,b\,:\,breezy-square\,.\,\Sigma\,c\,:\,gold-square\,.\,part(a,b,c)$
5. **pr5** $\triangleq$
   $\Sigma\,a\,:\,next-position-is-a-valid-square\,.\,\Sigma\,b\,:\,smelly-square\,.\,part(a,b)$
6. **pr6** $\triangleq$
   $\Sigma\,a\,:\,smelly-square\,.\,\Sigma\,b\,:\,gold-square\,.\,part(a,b)$

3.3.5 Liste des actions

Il existe 5 prédicats d'actions dont les types sont :

1. **go-home** $\triangleq\ agent\ \rightarrow\ nil$
2. **change-direction** $\triangleq\ agent\ \rightarrow\ position\ \rightarrow\ direction\ \rightarrow\ nil$
3. **go-forward** $\triangleq\ agent\ \rightarrow\ position\ \rightarrow\ direction\ \rightarrow\ nil$
4. **agent-keep-gold** $\triangleq\ agent\ \rightarrow\ gold\ \rightarrow\ nil$
5. **kill-Wumpus** $\triangleq\ agent\ \rightarrow\ position\ \rightarrow\ nil$

Et 7 types d'actions pour les Contextes (6 pour les Contextes agrégés qui ne sont pas présentés) :

1. **action1** $\triangleq$
   $\Sigma\,a\,:\sim agent-has-gold\,.\,change-direction(\pi_1(\pi_1 a)\pi_1(\pi_2(\pi_1 a))\pi_2({}_p i_2(\pi_1 a)))$
2. **action2** $\triangleq$
   $\Sigma\,a\,:\,smelly-square\,.\,change-direction(\pi_1(\pi_1(\pi_1 a))\pi_1(\pi_2(\pi_1(\pi_1 a)))\pi_2(\pi_2(\pi_1(\pi_1 a))))$
3. **action3** $\triangleq$
   $\Sigma\,a\,:\,breezy-square\,.\,change-direction(\pi_1(\pi_1(\pi_1 a))\pi_1(\pi_2(\pi_1(\pi_1 a)))\pi_2(\pi_2(\pi_1(\pi_1 a))))$

4. **action4** $\triangleq$
   $\Sigma\, a : next - position - is - a - valid - square\,.$
   $go - forward(\pi_1(\pi_1(\pi_1 a))\pi_1(\pi_2(\pi_1(\pi_1 a)))\pi_2(\pi_2(\pi_1(\pi_1 a))))$
5. **action5** $\triangleq$
   $\Sigma\, a : identify - dead - Wumpus - square\,.$
   $go{-}forward(\pi_1(\pi_1(\pi_1(\pi_1 a)))\pi_1(\pi_2(\pi_1(\pi_1(\pi_1 a))))\pi_1(\pi_2(\pi_1(\pi_1(\pi_1 a))))\pi_2(\pi_2(\pi_1(\pi_1(\pi_1 a)))))$
6. **action6** $\triangleq$
   $\Sigma\, a : identify - Wumpus - square\,.$
   $killed - the - Wumpus(\pi_1(\pi_1(\pi_1(\pi_1 a)))\pi_1(\pi_2(\pi_1(\pi_1(\pi_1 a)))))$
7. **action7** $\triangleq$
   $\Sigma\, a : gold - square\,.\, keep - gold(\pi_1(\pi_1(\pi_1 a))\pi_2 a)$

Comme indiqué dans l'introduction les contextes types sont au nombre de 9 (le contexte racine n'est pas pris en compte).

1. **Le contexte agent-location.**
   Ce contexte est une factorisation des informations contenues dans les contextes enfants *has-gold* et $\sim$ *has-gold*. Il permet de prouver qu'un agent existe bien dans le contexte en cours. Il permet également de vérifier que les propriétés position et direction de l'agent sont prouvées.
2. **Le contexte $\sim$ agent-has-gold.**
   Ce contexte est représenté par un $\Sigma - type$ emboîté qui associe un contexte agent-location à la proposition $\sim$ has-gold-p, qui permet de vérifier qu'il n'existe pas de relation entre l'agent en cours et de l'or (i.e. l'agent n'a pas d'or).
3. **Le contexte agent-has-gold.**
   Ce contexte est représenté par un $\Sigma - type$ emboîté qui associe un contexte agent-location à la proposition has-gold-p qui permet de vérifier qu'il existe bien une relation entre l'agent en cours et de l'or.
4. **Le contexte gold-square.**
   Ce contexte est une part-of de $\sim$ *agent-has-gold*, il indique que l'agent se trouve dans une case contenant l'or. En effet, la position en cours dégage un scintillement lui indiquant que le métal précieux se trouve dans cette case. La position en cours est enregistrée dans la base de connaissances lorsque l'agent se déplace. Toutes les informations concernant celle-ci sont alors disponibles. La relation entre l'or et cette case rend le prédicat *is-perceive-glitter-p* vérifié.
5. **Le contexte breezy-square.**
   La preuve du prédicat *is-breezy-position-p* valide le contexte et indique à l'agent qu'il se trouve dans une case adjacente à une case contenant un trou.
6. **Le contexte smelly-square.**
   Ce contexte se vérifie lorsque l'agent est adjacent à la case du Wumpus. Le prédicat *is-smelly-p* est alors démontré.
7. **Le contexte position-in-valid-square.**
   Dans ce cas l'agent se trouve sur une case qui ne possède aucune caractéristique propre (smelly ou breezy). Le prédicat *is-valid-position-p* est vérifié.

8. **Le contexte identify-Wumpus-square.**
   Ce contexte est beaucoup plus complexe que les Contextes précédents. En effet, il permet de déduire la case dans laquelle se trouve le Wumpus. De la connaissance de trois cases adjacentes à la case où se trouve le Wumpus, il est possible de déduire la position de cette case. Donc, le contexte smelly-square doit obligatoirement être vérifié avant de le valider (i.e. que le contexte est donc une part-of de smelly-square). Il suffit alors de trouver dans la base de connaissances deux positions différentes de la position en cours qui sont odorantes pour en conclure la position de la case contenant le Wumpus.
9. **Le contexte identify-dead-Wumpus-square.**
   Après la mort du Wumpus, les cases adjacentes à celle de la bête sont odorantes, cependant la case du Wumpus peut éventuellement être visitée par l'agent, puisque l'or pourrait éventuellement s'y trouver.

3.3.6 Exemple du contexte position-in-valid-square

L'explication du déroulement d'une étape de l'algorithme appliqué au choix d'un contexte puis au choix d'un contexte agrégé devrait suffire au lecteur pour comprendre le fonctionnement de l'algorithme.

Le contexte choisi est *next-position-valid-square*. La spécification du type de ce contexte correspond à un $\sum -type$ emboîté qui précise, par sous-typage, que le contexte parent $\sim$ *agent-has-gold* (i.e. que l'agent ne possède pas l'or, qui lui même est une extension du contexte *agent-location*), est adjacent à une case qui est sans risque.

L'expression du contexte par un $\sum -type$ est la suivante :

$$\Sigma\, a : \sim agent - has - gold\,.\, is - available - direction((\pi_1(\pi_2(\pi_1 a)))\,(\pi_2(\pi_2(\pi_1 a))))$$

L'agent peut être informé que la case suivante est sans danger par des informations qui lui sont fournies dans la base de connaissances. Ces informations peuvent être, par exemple, le fait que la case en cours n'est pas adjacente à celle du Wumpus (pas d'odeur), que la case n'est pas adjacente à celle d'un trou ou encore que l'agent est déjà passé par cette case.

La création du contexte passe, dans ce cas précis, par l'ajout d'une proposition portant sur le contexte parent qui indique que la case suivante est une case accessible à l'agent sans danger. Le type de cette proposition appelée *is-available-direction* est donné par le Π-type suivant :

$$(\Pi a : position.\Pi b : direction.available(a, b)).$$

Maintenant que l'agent a déterminé que la case suivante pouvait être explorée, il doit produire une action lui permettant de se déplacer dans cette case. Le contexte est donc associé à une fonction qui produit une action dont la spécification est donnée par le Σ-type :

$$\Sigma\, context\ :\ next - position - valid - square\,.$$
$$go - forward(\pi_1(\pi_1(\pi_1 context)))(\pi_1(\pi_2(\pi_1(\pi_1) context)))(\pi_2(\pi_2(\pi_1(\pi_1 context))))$$

La spécification du type dépendant *go-forward* est fournie par le Π-type suivant :

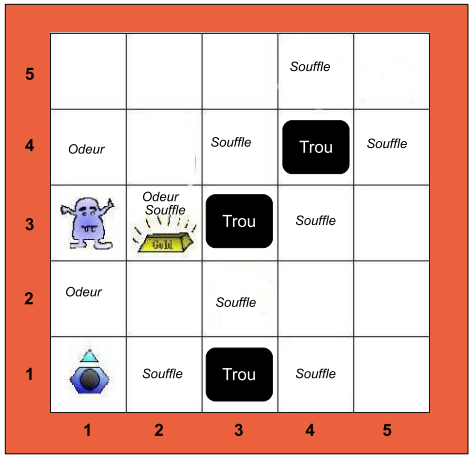
$$\Pi a : agent.\Pi b : position.\Pi c : direction.go(a, b, c)$$

Les arguments de l'action sont ainsi extraits du contexte valide et les éléments qui ne sont pas utilisés appartenant au contexte forment donc des contraintes sur ces arguments. La fonction instance du type *go* modifie la situation par des informations transmises aux actionneurs qui modifient la position de l'agent. Comme suite à cette action, les capteurs modifient la situation par une mise à jour des concepts qui composent la base de connaissances. Puis, le type action est ajouté à la base de connaissances (constituant un historique du système). Enfin, le cycle de recherche d'un contexte valide peut reprendre et ainsi de suite.

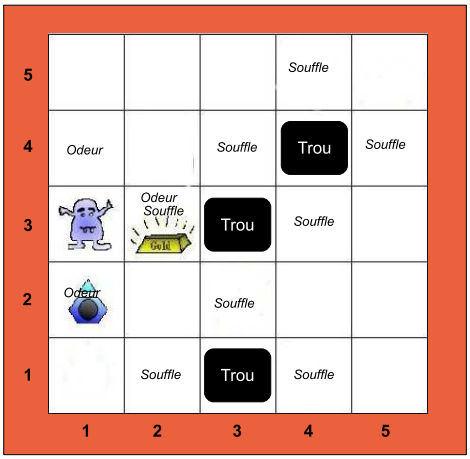
Voici maintenant l'étude d'un cas dans lequel un contexte agrégé est vérifié. Pour cela, il est supposé comme montré à la figure 3.3.6 que l'agent se trouve dans une case contenant de l'or. Elle dégage une odeur (la case est adjacente à celle du Wumpus) et elle est adjacente à un trou. Les trois contextes breezy-square, smelly-square et gold-square sont alors vérifiés. L'algorithme peut démontrer qu'un ou plusieurs contextes agrégés sont vérifiés. Le Contexte agrégé le plus précis *pr*4 est alors validé. C'est donc l'action qui est associée à ce contexte (i.e. l'action 7) qui est exécutée.

Les figures suivantes présentent le résultat imagé de l'exécution de la résolution d'une grille 5x5 du Wumpus présenté dans le livre de Norvig et Russel. La trace d'exécution est fournie en annexe A.

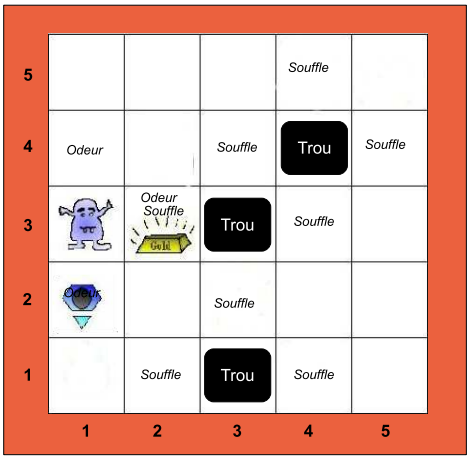
Il faut noter que toutes les opérations de changement de *direction* ne sont pas représentées sur les graphiques.



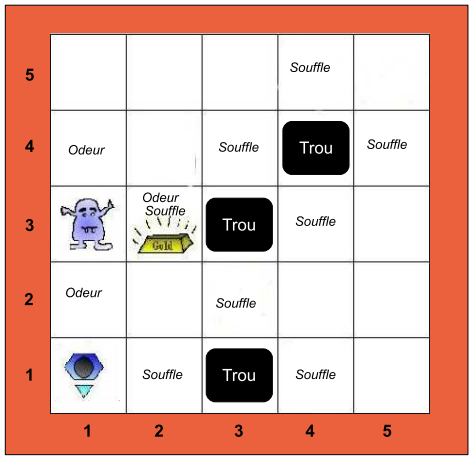
Etape 1 :
Contexte vérifié :
agent-location $\triangleq \Sigma a : agent - Wumpus.\Sigma b : agent - position(a).agent - direction(a)$
Action :
go-forward $\triangleq agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 2 :
Contexte vérifié :
smelly-square $\triangleq$
$\Sigma a :\ agent - has - gold.is - smelly - position - p(\pi_1(\pi_2(\pi_1 a)))$
Action :
change-direction $\triangleq agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 3 :
Contexte vérifié :
next-position-is-a-valid-square $\triangleq$
$\Sigma a :\ agent - has - gold.is - possible - direction(\pi_1(\pi_2(\pi_1 a))\pi_2(\pi_2(\pi_1 a)))$
Action :
go-forward $\triangleq agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 4 :

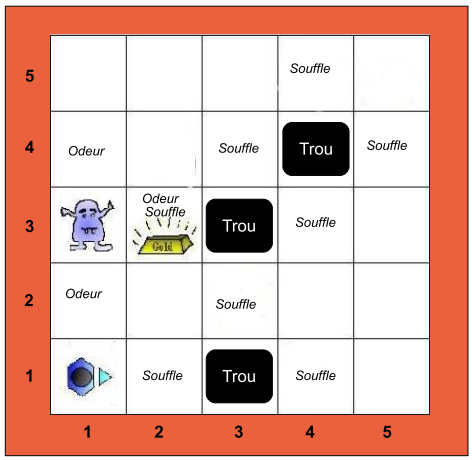
Contexte vérifié :

agent-has-gold $\triangleq$

$\Sigma a : agent - location.\ has - gold - p(\pi_1 a)$

Action :

change-direction $\triangleq$ $agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 5 :

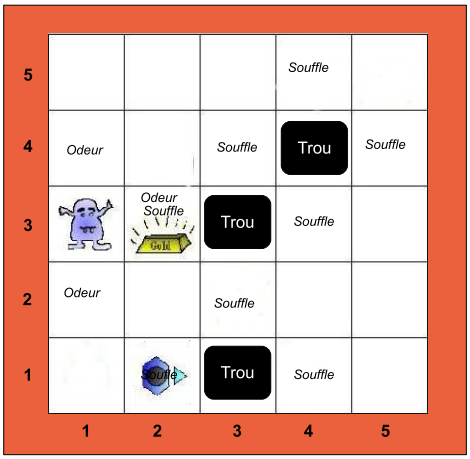
Contexte vérifié :

next-position-is-a-valid-square $\triangleq$

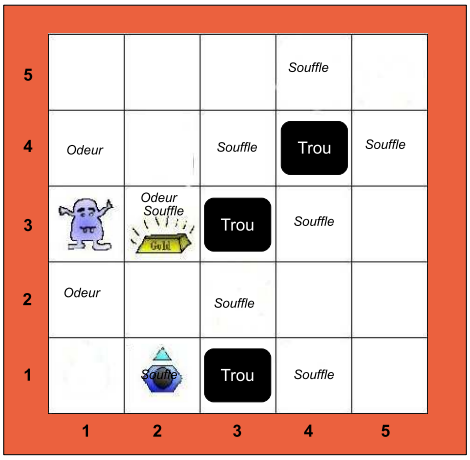
$\Sigma a :\ agent - has - gold.is - possible - direction(\pi_1(\pi_2(\pi_1 a))\pi_2(\pi_2(\pi_1 a)))$

Action :

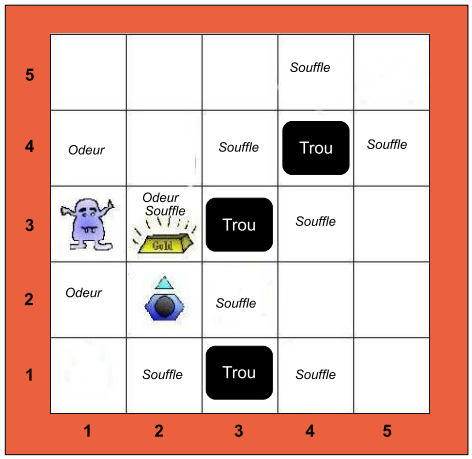
go-forward $\triangleq$ $agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 6 :
Contexte vérifié :
breezy-square $\triangleq$
$\Sigma a :\ agent - has - gold.is - breezy - position - p(\pi_1(\pi_2(\pi_1 a)))$
Action :
change-direction $\triangleq agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 7 :
Contexte vérifié :
next-position-is-a-valid-square $\triangleq$
$\Sigma a :\ agent - has - gold.is - possible - direction(\pi_1(\pi_2(\pi_1 a))\pi_2(\pi_2(\pi_1 a)))$
Action :
go-forward $\triangleq agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 8 :

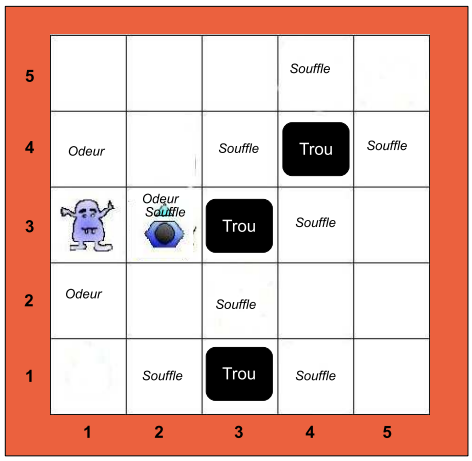
Contexte vérifié :

next-position-is-a-valid-square $\triangleq$

$\Sigma a :\ agent - has - gold.is - possible - direction(\pi_1(\pi_2(\pi_1 a))\pi_2(\pi_2(\pi_1 a)))$

Action :

go-forward $\triangleq agent \rightarrow position \rightarrow direction \rightarrow nil$



Etape 9 :

Contexte vérifié :

pr4 $\triangleq$

$\Sigma a : smelly - square.\Sigma b : breezy - square.\Sigma c : gold - square.part(a, b, c)$

Action :

action7 $\triangleq$

$\Sigma a : gold - square.keep - gold(\pi_1(\pi_1(\pi_1 a))\pi_2 a)$

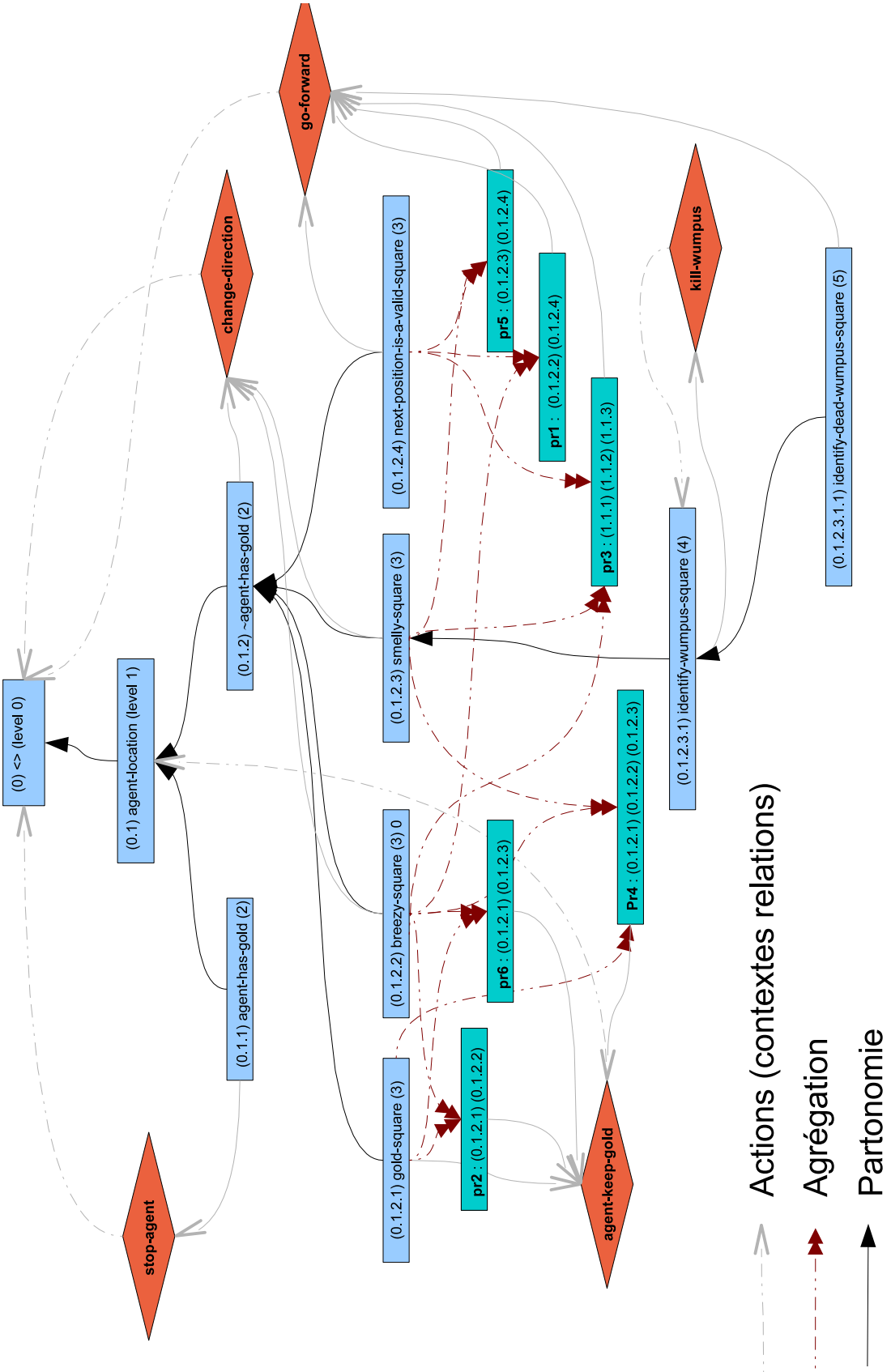


FIGURE 3.1 – Les Contextes du Wumpus

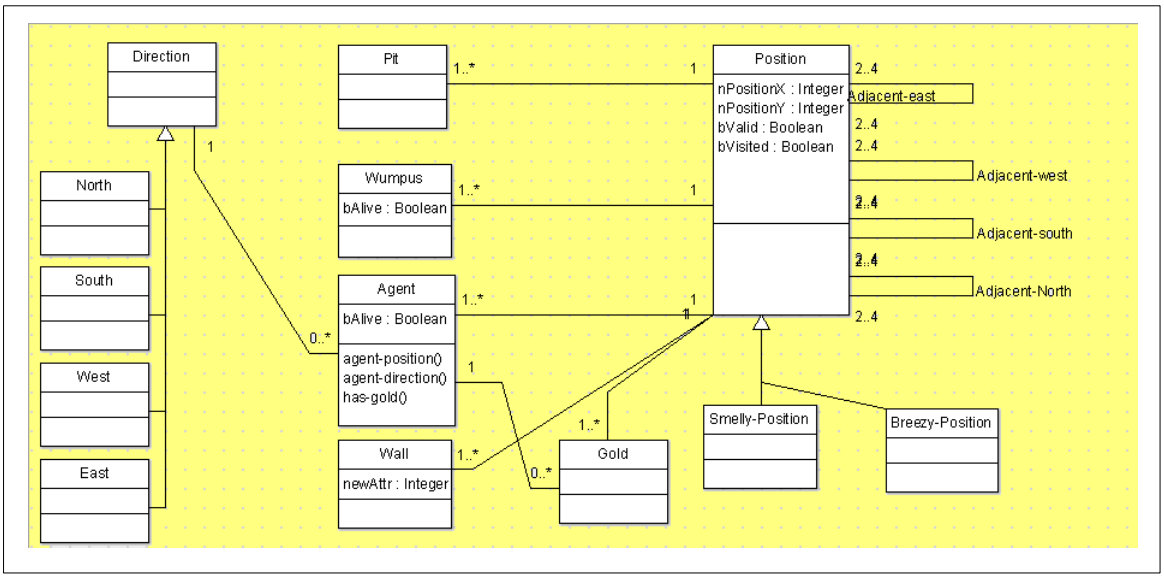


FIGURE 3.2 – Les concepts du Wumpus

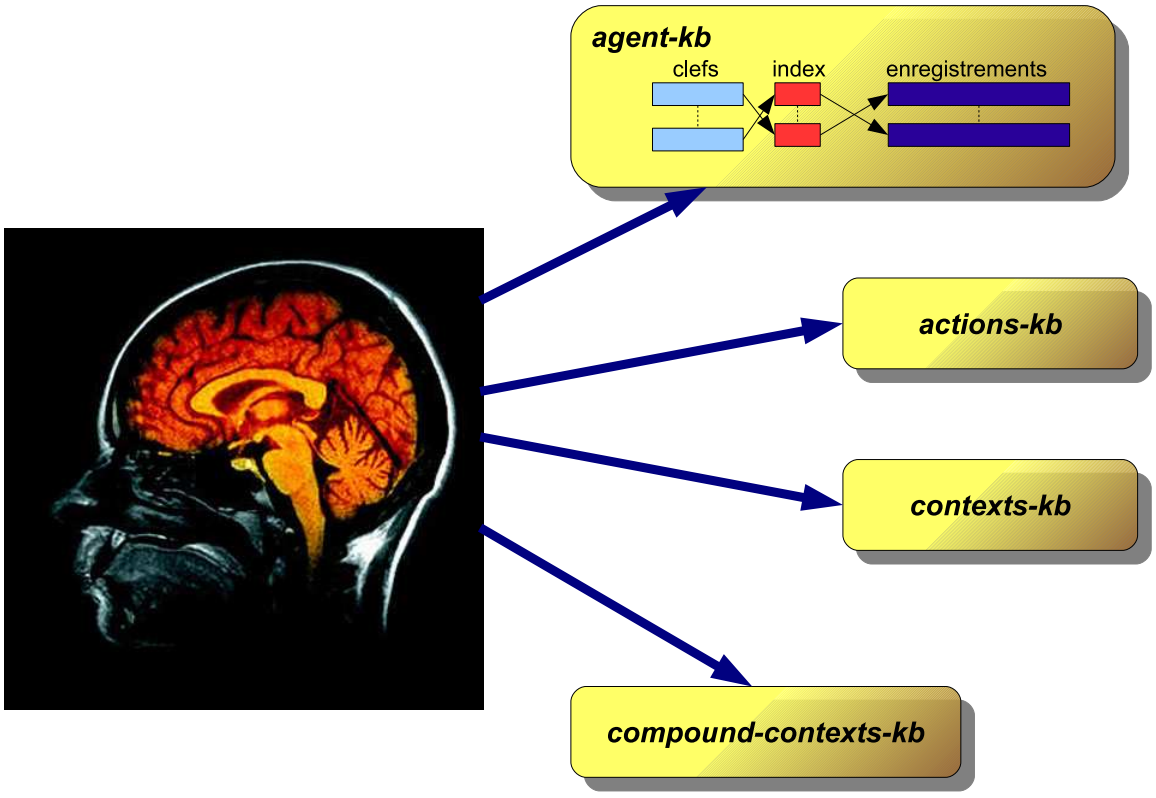


FIGURE 3.3 – Structure de la base de connaissances d'un agent

```
(let ((name 'breezy-square-change-direction)
        (lst-types '(breezy-square))
        (subsumed '())
        (subsuming '())
        (meronym '())
        (holonym '())
        (level nil)
        (constructor  '(lambda (x)
                         (if (typep x 'breezy-square)
                             (let*((a x)
                                   (b (change-direction (pi-1 (pi-1(pi-1 a)))
                                                        (pi-1(pi-2 (pi-1(pi-1  a))))
                                                        (pi-2(pi-2 (pi-1(pi-1  a))))
                                                        )))
                               (progn
                                 (if  (not(equal b nil))
                                      (list a b)
                                      ))
                               )
                             )
                         )))
  (defun get-level-breezy-square-change-direction()
    level
    )
  (defun initialise-breezy-square-change-direction()
    (setf current-value '())
    )
  (defun get-subsumed-breezy-square-change-direction ()
    subsumed
    )
  (defun get-subsuming-breezy-square-change-direction  ()
    subsuming
    )
  (defun get-meronym-breezy-square-change-direction  ()
    meronym
    )
  (defun get-holonym-breezy-square-change-direction  ()
    holonym
    )
  (defun proof-breezy-square-change-direction ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-breezy-square-change-direction ()
    lst-types
    )
  (defun  breezy-square-change-direction (breezy-square)
    (funcall (eval constructor ) breezy-square)))
```

FIGURE 3.4 – Exemple de code en LISP d'une action

4

Etude de la complexité

Soit la donnée des variables $\in N^*$ suivantes :

1. m_1 représente le nombre d'arcs dans le graphe des Contextes.
2. n_1 représente le nombre de sommets dans le graphe des Contextes.
3. q représente le nombre des types simples.
4. s représente le nombre des $\Sigma - types$.
5. p représente le nombre des $\Pi - types$.

Le calcul de la complexité passe par la somme des itérations de la boucle n, soit :

$$\sum_{i=1}^{i=n} Proof_i$$

où $Proof_i$ est l'évaluation de l'algorithme 4.

Complexité des algorithmes :

Les règles de construction du graphe d'exécution, qui correspondent à un ordonnancement de l'évaluation des types de Contextes, permettront, lors de la recherche d'objets instances du Contexte dans la situation courante, de diminuer la complexité de l'algorithme. Il aurait été possible de rechercher dans la base de connaissances les instances des Types arguments du Contexte, puis de calculer le produit cartésien de ces listes, ce qui aurait porté la complexité au niveau de la classe des algorithmes exponentielles ($\mathcal{O}(m^n)$). Au lieu de cela, la préférence fût donnée à un algorithme travaillant sur des sélections (les $\Pi - types$) et sur les intersections (les $\Sigma - types$), ceci évidemment pour ramener la complexité de l'algorithme à une classe de complexité quadratique ($\mathcal{O}(n^2)$).

Pour faire baisser la complexité de l'algorithme, c'est à la création des Contextes types que le programme génère automatiquement un graphe d'exécution des différents Types, dépendants ou non, sources des Contextes types. Ainsi, lors de l'évaluation de celui-ci par l'automate, il lui suffira de parcourir les noeuds du graphe jusqu'au dernier sommet, appelé sommet résultat. Le comportement de l'algorithme se rapproche alors de celui d'un automate utilisé dans les calculs de recherche de sous-chaînes de caratères [5]. Lorsqu'un sommet du graphe ne peut pas être

vérifié, l'algorithme s'arrête.

Soit un arbre de partonomie appelé G_1, dont les noeuds sont les contextes, $n1$ le nombre de noeuds et m_1 le nombre d'arcs de ce graphe. Chaque sommet x_i (i.e. un contexte) est lui même un graphe G_2 contenant n_2 sommets et m_2 arcs. Le calcul de la complexité C_1 totale du démonstrateur de DTF peut donc s'exprimer en tenant compte de deux parties $C(G_1)$ et $C(G_2)$ telles que $C_1 = C(G_1)C(G_2)$ avec $C(G_2)$ correspondant à la complexité de l'algorithme 2 et $C(G_1)$ à celle de l'algorithme 1.

La recherche des objets instances d'un type (i.e. le type inhabitation problem) de Contexte, (i.e. le graphe G_2) demande la résolution de trois algorithmes :

1. Le produit cartésien (classe de complexité quadratique $\mathcal{O}(n^2)$).
2. Une opération qui peut se rapprocher de l'équi-jointure de l'algèbre relationnelle, i.e. des éléments de deux tuples donnés par des $\sigma - types$ sont dérivés pour former un nouveau tuple. ($\mathcal{O}(nm)$).
3. Une opération semblable à l'opération de sélection de l'algèbre relationnelle (réalisée par des $\Pi - types$).

Avec α, β et γ les nombres respectifs des sommets du graphe correspondant aux types précédants avec $G_2(\alpha + \beta + \gamma = n_2)$.

Soit k le nombre de preuves d'objets prouvés de l'application, la complexité moyenne du graphe G_2 peut s'écrire de la manière suivante $\alpha k^2 + \beta k k' + \gamma k$. Un majorant de ce polynôme est $(\alpha + \beta)k^2$, lui même majoré par la fonction $n_2 k^2$. Le graphe G_1 est étendu par les contextes agrégés (algorithme 2). Cet algorithme permet de calculer les preuves des contextes agrégés et possède un espace de recherche limité aux contextes prouvés par l'algorithme 1, lui-même borné par n_1.

La complexité de l'algorithme qui calcule les preuves des Contextes est en $C'_T = (n_1 + C(G_1))c(G_2)$. L'algorithme 3 recherche la preuve d'un Contexte dans le graphe G_1 qui est borné par $(n_1, m_1) \leq n_1$. L'algorithme 4 nécessite une recherche des preuves des actions égales au nombre de Contextes prouvés soit k. Finalement, la complexité de l'algorithme est égale à : $C'_T = 2k + (2n_1)k^2 n_2$ (le signe $+$ indique les séquences entre les algorithmes 3 et 4).

En pratique, il est possible de distinguer deux cas :

i Peu de Contextes et un grand nombre k de preuves, alors la complexité est en $\mathcal{O}(k^2)$.

ii Beaucoup de Contextes avec un petit nombre de preuves, alors la complexité est en $\mathcal{O}(n_1 n_2)$.

Cela entraîne la conclusion suivante :

La complexité est quadratique dans le cas le plus défavorable.

5

Conclusion

L'expérience montra qu'il est tout à fait envisageable de réaliser une implantation efficace d'un agent DTF en Lisp et de tester la faisabilité de la représentation Contextes Actions par un test de gestion d'un agent autonome devant se déplacer dans un environnement inconnu. Actuellement, des développements sont à l'étude par EDF pour tester DTF dans le cadre de la gestion d'un appartement intelligent (e.g., aide aux personnes agées).

Cependant, à la lumière de cette simulation, plusieurs zones d'ombres subsistent, en effet, si l'expressivité de DTF permet de limiter les prédicats nécessaires à la représentation de contraintes et autres propriétés plus complexes du monde, la modélisation des Contextes reste du ressort d'un spécialiste du domaine et peut éventuellement être plus difficile à réaliser que des représentations du contexte plus connues comme les graphes conceptuels emboîtés de Sowa. Ces derniers pourraient donc être utilisés comme des représentations de base du domaine, puis être convertis en une expression DTF des Contextes sans trop de difficultés.

Il existe d'autres solutions, comme indiqué en début de partie. Ainsi, il est également envisagé d'étudier une solution à partir de textes du domaine (p. ex., le Corpus). Le fait d'utiliser le langage Lisp pour un prototype n'est qu'une option qui limite et simplifie grandement le codage du démonstrateur. Mais il est tout à fait possible d'utiliser, pour coder le démonstrateur, un langage objet comme Java. Il reste que le code sera plus long et certainement moins performant en terme de vitesse, mais cela reste à confirmer. Comme indiqué lors de la conception de l'exemple du Wumpus, une première implémentation du Wumpus a tout d'abord été réalisée. Elle utilisait le langage RDF (en Lisp) pour enregistrer les Contextes et des requêtes SPARQL sur la base pour prouver les Contextes et les actions. Mais cette solution a été abandonnée, car les résultats en terme de vitesse d'exécution n'étaient pas probants.

Par contre la représentation des Contextes et des Actions peut relativement aisément être sérialisée en RDF en utilisant XML (e.g., Xerces) ou pour un autre schéma, pour des échanges ou pour la réalisation des traces d'exécution à des fins de conception d'un historique etc.

5.1 Analyse des résultats

Cette dernière section dresse un bilan des apports de ce travail de recherche et donne également quelques perspectives.

5.1.1 Principaux apports

Dans ce mémoire, il a été montré comment les mathématiques constructives, et plus précisément la logique intuitionniste, au travers de la théorie des types dépendants d'ordre supérieur, pouvaient être utilisées, par la correspondance de Curry-Howard, comme un outil efficace pour la définition d'applications pour la représentation de connaissances partielles.

Le cadriciel proposé, qui porte le nom de DTF, a permis d'amener une solution à des problèmes rencontrés par les langages actuels à savoir :

- L'expressivité.
- L'évolutivité.
- L'optimisation par diminution de l'espace de recherche.
- La décidabilité.

La Théorie des Types représente les preuves par des programmes dans un langage simple, basé sur le λ-calcul, et les propriétés par les types de ces programmes. Pour pouvoir manipuler des propriétés et des spécifications plus complexes, il existe des types - appelés types dépendants - qui dépendent de termes du langage (termes qui peuvent être des valeurs).

Les types dépendants augmentent très nettement le pouvoir d'expressivité de la théorie des types. Ce pouvoir d'expressivité a été employé pour exprimer trois niveaux de représentations : le niveau logique, le niveau épistémologique et le niveau ontologique. La syntaxe qui est commune aux trois niveaux garantit une homogénéité du modèle simplifiant l'échange d'informations entre les couches et permettant de réaliser des inférences logiques au niveau épistémologique et ontologique.

Par ailleurs, il est possible de définir une structure équivalente aux métaclasses dans lesquelles des classes peuvent être considérées comme des objets. Des attributs possèdent la qualité d'être polymorphes simplement en les introduisant dans des variables liées de types somme différents. Il est également possible de pré-définir des contraintes, ainsi que des valeurs numériques, des comparaisons, etc., au sein de types somme. Des relations "n-aires" sont formalisables par des types somme emboîtés de profondeur quelconque. Le typage fournit des contraintes sur le type des arguments, qui peuvent être utilisées dans le cadre de spécifications de contraintes d'intégrité. Le système de typage rend simultanément possible la spécification d'instances de concepts et d'instances de relations. Enfin, les mécanismes d'inférences d'ordre supérieur placent DTF au delà des systèmes classiques de type DL.

Avec DTF, l'exploitation de l'adéquation qui existe entre d'une part le système de typage dépendant et les ontologies, et d'autre part le système de typage et les programmes permet de garantir la décidabilité par les programmes.

Le choix des représentations ontologique et épistémologique assure à la solution des perspectives d'évolution dynamique de ces différents modèles, au niveau ontologique par l'ajout en extension d'informations (concepts ou relations), et au niveau épistémologique par l'ajout de sous-types des contextes courants, voire l'ajout d'un nouveau domaine de contextes.

Le domaine contextuel (i .e., l'arbre de subsomption) d'une application est construit par des contextes, polysémies d'un contexte de base (par raffinement du contexte de base). En effet, tous les sous-contextes du contexte racine d'un domaine d'application ajoutent de la précision au contexte inférieur et ainsi de suite. Ce principe de découpage de l'information réduit donc l'espace de recherche et augmente d'autant la performance de l'algorithme qui, dans le pire des cas, est évaluée avec une complexité quadratique.

5.1.2 Perspectives

Si les types peuvent dépendre des valeurs du système, deux types peuvent être équivalents tout en étant syntaxiquement différents. Par exemple, les listes de longueur 1 + 1 sont exactement les mêmes que les listes de longueur 2. Pour qu'un système puisse détecter le plus possible de ces équivalences, il est nécessaire d'implanter un procédé appellé la coercion.

L'implantation de la coercion avec des types dépendants par des fonctions est à l'étude pour DTF. Mais, si la conversion des types simples est un sujet connu qui ne pose pas de problème, il en est tout autrement de la conversion des types dépendants qui, elle, nécessite la définition d'un algorithme de résolution beaucoup plus complexe. En effet, les relations de sous-typage pour les types dépendants incluent des relations d'équivalence de types. Une représentation explicite de cette relation est donnée par la règle suivante : si X et Y sont convertibles, alors X est un sous-type de Y.

La conversion de types permet une augmentation significative du nombre des Contextes que l'algorithme aura la possibilité de prouver. En effet, la substitution de types par d'autres, lors de la démonstration des Contextes, permettra évidemment d'identifier un plus grand nombre de Contextes pour une situation donnée, mais également d'envisager le partage des traitements entre des agents.

L'algorithme DTF-A est conçu pour traiter des descriptions écrites en LISP, ce qui rend difficile son écriture par des personnes non qualifiées. Une étude est donc en cours pour réaliser une description standard par l'intermédiaire du langage XML de la définition des Contextes et des Actions décritent avec un outil graphique. Il deviendra alors posssible de générer des contextes depuis des applications existantes comme des interfaces graphiques de conception ou d'autres, mais également d'utiliser ce formalisme pour l'échange de Contextes entre les agents.

Comme la théorie est stratifiée (sortes, types et objets), elle peut être utilisée pour la spécification de systèmes et pour prouver qu'ils respectent un ensemble de règles, i.e. qu'il est possible de faire ce que les essentialistes appellent du méta, sans sortir de la théorie.

Pour finir, si DTF est destiné à des applications à environnement changeant, son utilisation pourra également être étendue à d'autres applications, comme par exemple la spécification d'ontologie, ou encore le Web-sémantique, à ce propos nous pouvons reprendre le propos de Tim Berners-Lee : "if there is a semantic web machine, then it is a proof validator, not a theorem

prover" [Tim Berners-Lee].

N.B. : Ce travail a donné lieu à de nombreuses publications dans des conférences internationales, contributions à des chapitres d'ouvrages et un revue nationale [47], [46], [10], [43], [44], [10], [42], [9], [41], [41],[45].

A

Trace de l'exécution du Wumpus

```
* (proof-automath)
2
"lst-of-deepest-context"
4 ((NEXT-POSITION-IS-A-VALID-SQUARE
    ((((#<AGENT-WUMPUS {405DE315}> (#<POSITION {4056AC65}> #<NORTH {4056910D}>))
6       T)
      T))))
8 "inference-forward-position"
"go-forward"
10 1
"Direction : "
12 #<NORTH {4056910D}>
"Position X"
14 1
"Position Y"
16 2
"lst-of-deepest-context"
18 ((SMELLY-SQUARE
    ((((#<AGENT-WUMPUS {405DE315}> (#<POSITION {4056BD6D}> #<NORTH {4056910D}>))
20      T)
      T))))
22 "change-direction"
(#<SOUTH {40569C75}>)
24 2
"Direction : "
26 #<SOUTH {40569C75}>
"Position X"
28 1
"Position Y"
30 2
"lst-of-deepest-context"
32 ((NEXT-POSITION-IS-A-VALID-SQUARE
    ((((#<AGENT-WUMPUS {405DE315}> (#<POSITION {4056BD6D}> #<SOUTH {40569C75}>))
34      T)
      T)))
36  (SMELLY-SQUARE
    ((((#<AGENT-WUMPUS {405DE315}> (#<POSITION {4056BD6D}> #<SOUTH {40569C75}>))
38      T)
      T))))
40 "inference-forward-position"
"go-forward"
42 3
"Direction : "
44 #<SOUTH {40569C75}>
"Position X"
46 1
"Position Y"
48 1
"lst-of-deepest-context"
50 ((NEXT-POSITION-IS-A-VALID-SQUARE
    ((((#<AGENT-WUMPUS {405DE315}> (#<POSITION {4056AC65}> #<SOUTH {40569C75}>))
52      T)
      T))))
54 "inference-forward-position"
"go-forward"
56 4
"Direction : "
58 #<SOUTH {40BB87A5}>
"Position X"
60 1
"Position Y"
62 1
"lst-of-deepest-context"
64 ((¯AGENT-HAS-GOLD
    (((#<AGENT-WUMPUS {40BC496D}> (#<POSITION {40BA0BAD}> #<SOUTH {40BB87A5}>))
66      T))))
"change-direction"
68 NIL
5
70 "Direction : "
#<EAST {40BB8715}>
72 "Position X"
1
74 "Position Y"
1
76 "lst-of-deepest-context"
((NEXT-POSITION-IS-A-VALID-SQUARE
78    ((((#<AGENT-WUMPUS {40BC496D}> (#<POSITION {40BA0BAD}> #<EAST {40BB8715}>))
      T)
80    T))))
"inference-forward-position"
82 "go-forward"
"inference-forward-position"
84 6
"Direction : "
86 #<EAST {40BB8715}>
"Position X"
88 2
"Position Y"
90 1
"lst-of-deepest-context"
92 ((BREEZE-SQUARE
```

```
    ((((#<AGENT-WUMPUS {40BC496D}> (#<POSITION {40BC4795}> #<EAST {40BB8715}>))
94        T)
       T))))
96  "change-direction"
    NIL
98  "inference-forward-position"
    7
100 "Direction : "
    #<NORTH {40BB8775}>
102 "Position X"
    2
104 "Position Y"
    1
106 "lst-of-deepest-context"
    ((NEXT-POSITION-IS-A-VALID-SQUARE
108   ((((#<AGENT-WUMPUS {40BC496D}> (#<POSITION {40BC4795}> #<NORTH {40BB8775}>))
          T)
110       T)))
      (BREEZE-SQUARE
112   ((((#<AGENT-WUMPUS {40BC496D}> (#<POSITION {40BC4795}> #<NORTH {40BB8775}>))
          T)
114       T))))
    "inference-forward-position"
116 "go-forward"
    "inference-forward-position"
118 8
    "Direction : "
120 #<NORTH {40FB3C35}>
    "Position X"
122 2
    "Position Y"
124 2
    "lst-of-deepest-context"
126 ((NEXT-POSITION-IS-A-VALID-SQUARE
    ((((#<AGENT-WUMPUS {40FBF9C5}> (#<POSITION {40F970E5}> #<NORTH {40FB3C35}>))
128       T)
       T))))
130 "inference-forward-position"
    "go-forward"
132 "inference-forward-position"
    9
134 "Direction : "
    #<NORTH {40FB3C35}>
136 "Position X"
    2
138 "Position Y"
    3
140 "lst-of-deepest-context"
    ((BREEZE-SQUARE
142   ((((#<AGENT-WUMPUS {40FBF9C5}> (#<POSITION {40FBF745}> #<NORTH {40FB3C35}>))
          T)
144       T)))
      (GOLD-SQUARE
146   ((((#<AGENT-WUMPUS {40FBF9C5}> (#<POSITION {40FBF745}> #<NORTH {40FB3C35}>))
          T)
148       #<GOLD {40FBF9A5}>)))
      (SMELLY-SQUARE
150   ((((#<AGENT-WUMPUS {40FBF9C5}> (#<POSITION {40FBF745}> #<NORTH {40FB3C35}>))
          T)
152       T))))
    " keep-gold"
154 "inference-forward-position"
    10
156 "Direction : "
    #<NORTH {40FB3C35}>
158 "Position X"
    2
160 "Position Y"
    3
162 "lst-of-deepest-context"
    ((AGENT-HAS-GOLD
164   (((#<AGENT-WUMPUS {40FBF9C5}> (#<POSITION {40FBF745}> #<NORTH {40FB3C35}>))
       T))))
166 " go-home"
    "inference-forward-position"
168 11
    "Direction : "
170 #<NORTH {40FB3C35}>
    "Position X"
172 2
    "Position Y"
174 3
    NIL
```

B

Code Lisp de DTF-A

```
;;;; Created on 2008-10-17 17:23:38
  ;;; Décider du niveau.
;;;;(load "/home/patrick/Thesis/IntContextType/v02a/IntContextType002/IntContextType002/Proof/p-proof-algorithm-of-t
(let ((lst-of-current-valid-context '())
        (lst-of-deepest-context '())
        (lst-of-compound '())
        (active-agent t)
        (current-type 'root)
        (current-goal nil)
        (sigma-types-extension (make-hash-table))
      )
   (defun add-sigma-types-extension (name extension)
     (setf (gethash name sigma-types-extension)  extension)
  )
  (defun get-extension-of-sigma-types(sigma-name)
    (let ((sg-name))
      (if (typep sigma-name 'compound-types)
          (setf sg-name (gethash sigma-name agent-compound-types-kb))
          (setf sg-name sigma-name)
      )
      (gethash sg-name sigma-types-extension)
    ))
  (defun initialise-extension-of-sigma(name)
     (setf (gethash name sigma-types-extension)  '())
    )
  (defun sigma-types-p (value)
    (gethash value sigma-types-extension)
    )
 (defun is-extension(sigma-name)
   (not (equal (gethash sigma-name sigma-types-extension) nil))
  )
  (defun set-current-goal (goal-name)
    (setf current-goal goal-name)
    )
  (defun start-agent ()
    (if current-goal
        (progn
          (setf active-agent t)
          (proof-automath))
        (print "No current-goal defined for this agent ! Please give a goal."))
    )
  (defun get-current-type()
    current-type
    )
  (defun set-current-type(c-type)
    (setf current-type c-type)
    )
  (defun set-current-type-if-low-level (c-type)
    (let ((current-level (get-level-of-a-type current-type))
          (new-level (get-level-of-a-type c-type)))
      (if (< new-level current-level)
          (set-current-type c-type)
          )
      )
    )
  (defun set-true-active-agent()
    (setf active-agent t)
    )
  (defun set-stop-active-agent()
    (setf active-agent nil)
    )
  (defun get-lst-of-compound ()
   lst-of-compound
   )
 (defun get-lst-of-deepest-context ()
   lst-of-deepest-context
```

```
67    )
 (defun set-lst-of-deepest-context  (lst)
69    (setf lst-of-deepest-context lst))

71   (defun set-lst-of-current-valid-context (lst)
      (setf lst-of-current-valid-context lst))
73
     (defun get-lst-of-current-valid-context ()
75     lst-of-current-valid-context
       )
77 ;;;Les contextes sont enregistrés avec le niveau auquel ils sont liés.
    (defun get-proofs-of-the-current-situation (context-root level)
79   ;;;Le contexte racine (context-root) est déjà prouvé.
     ;;;Le contexte racine en cours est un sigma-type.
81     (let* ((local-level (+ level 1 ))
                (lst-of-root-childs (get-lst-of-subsuming-element-of-a-type context-root))
83       ;;;Parcourir la liste en recherchant une preuve pour chaque DRT.
                (lst-of-proof-root-childs (mapcar #'(lambda (x) (get-lst-proof-of-tdr x)) lst-of-root-childs))
85              (lst-of-proof-root-childs-with-type (mapcar #'(lambda (x y) (if (not(equal y nil))
                                                                                (progn
87                                                                                (add-sigma-types-extension x y)
                                                                                  (list x y))))
89                                                                  lst-of-root-childs lst-of-proof-root-childs))
                (lst-temp '())
91            )
        (progn
93        (setf lst-of-proof-root-childs-with-type (remove-nil lst-of-proof-root-childs-with-type))
          (setf lst-temp lst-of-proof-root-childs-with-type)
95
          ;;(print "lst-of-root-childs")
97        ;;(print lst-of-root-childs)
          ;;(print "Current level")
99        ;;(print local-level)
          ;;(print lst-of-proof-root-childs-with-type )
101       (if (not (equal lst-temp nil))
              (progn
103             ;;(print "lst-temp")
                ;;(print lst-temp)
105             (pushnew (pushnew level lst-temp) lst-of-current-valid-context)
                (loop for z in lst-of-proof-root-childs-with-type do
107                 ;;;Ne prendre en compte que les types dont une preuve a pu être démontrée.
                    (get-proofs-of-the-current-situation (first z) local-level)
109                 )
            ))
111       )
        )
113   )
;;;Example :
115  ;; (add-sigma-types-extension 'agent-location (get-lst-proof-of-tdr 'agent-location));;Ok
     ;;  (get-proofs-of-the-current-situation 'agent-location 1);;Ok
117  ;;  (get-lst-of-current-valid-context);;Ok
;;;;Déterminer la liste des DTR de niveau le plus bas. Ok
119
 (defun set-lst-of-the-deepest-dtr()
121   (let ((level (get-valid-level lst-of-current-valid-context))
            (lst '()))
123     (progn
       (loop for x in lst-of-current-valid-context do
125           (progn
             (setq current-level  (first x))
127          (if (equal level current-level)
                (setq lst (append (cdr x) lst))
129             )))
       )
131    (setq lst-of-deepest-context lst)
       )
133  )
 ;;;Example
135 ;;(set-lst-of-current-valid-context ;;;Ok
;;  '((1 agent-location (1 2)) (2 (~agent-has-gold (1 2)) ) (3 (breeze-square (1 2))) (3 (gold-square (5 4)))
(3 (smelly-square (6 7)))));;;Ok
137 ;;(set-lst-of-the-deepest-dtr);;;Ok
;;(get-lst-of-deepest-context);;;Ok
139 ;;;Rechercher les contraintes... Ok
   (defun get-proofs-of-current-compound-dtr ( )
141   (let* ((lst-of-constraints-context
                 (get-lst-of-drt-of-constraints (eliminated-constraints-double
143                (get-lst-of-all-meronym (get-lst-type lst-of-deepest-context)))))
              (lst-of-valid-constraints (eliminated-no-valid-constraints
145                                                                lst-of-constraints-context lst-of-deepest-context)
147           )
      lst-of-valid-constraints
149   ))
 ;;;Example :
```

```
;;(setf lst-of-cvc '((breeze-square (1 2)) (gold-square (5 4)) (smelly-square (6 7))))
;;(get-lst-type lst-of-cvc) ;;Ok
;;(get-lst-of-all-meronym (get-lst-type lst-of-cvc)) ;;Ok
;;(setf lst-of-constraints-context
;;      (get-lst-of-drt-of-constraints
;;       (eliminated-constraints-double (get-lst-of-all-meronym (get-lst-type lst-of-cvc)))));;Ok
;;(eliminated-no-valid-constraints lst-of-constraints-context lst-of-cvc);;Ok
;; (set-lst-of-deepest-context '((breeze-square (1 2)) (gold-square (5 4)) (smelly-square (6 7))));;Ok
;;  (get-proofs-of-current-compound-dtr);;;Ok
 (defun execute-actions ()
   (let ((lst))
     (progn
     (if (equal nil lst-of-compound)
         (setf lst lst-of-deepest-context)
         (setf lst lst-of-compound )
         )
     ;;(print lst)
     (mapcar #'(lambda (x)
                 (execute-action (first x)))  lst)
       ))
   )

 (defun erase-proof ()
   (del-all-son-of-dtr current-type)
   (setf lst-of-deepest-context nil)
   (setf lst-of-current-valid-context nil)
   )

 (defun execute-deductions ()
   (let ((lst (gethash 'deductions-lst agent-kb)))
     (mapcar #'(lambda (x)
                 (get-lst-proof-of-tdr x))   lst)
     )
   )
(defun is-goal-valid()
  (let ((goal nil))
    (mapcar #'(lambda (x)
                (if (equal (first (first (cdr x))) current-goal)
                    (setf goal current-goal)
                    ))   lst-of-current-valid-context)
    goal
    )
  )
 (defun proof-automath()
   (let ((i 0))
   (loop while (equal active-agent t)
         do(progn
            (setf i (+ i 1))
            (get-proofs-of-the-current-situation current-type 1)
             (if (is-goal-valid)
                 (progn
                   (print "Ok goal !")
                   (set-stop-active-agent))
                 (progn
                   (set-lst-of-the-deepest-dtr)
                   (print "lst-of-deepest-context")
                   (print lst-of-deepest-context)
                   (setf lst-of-compound (get-proofs-of-current-compound-dtr))
                   ;;(print "lst-of-compound")
                   ;;(print lst-of-compound)
                   (execute-actions)
                   (execute-deductions)
                   (erase-proof)
                   ;;(if (= 10  i) (set-stop-active-agent))
                   (print i)
                   (print "Direction : ")
                   (print (direction current-agent))
                   (print "Position X")
                   (print (coordinateX (positionG current-agent)))
                   (print "Position Y")
                   (print (coordinateY (positionG current-agent)))))
              )
       ))
   )

 ;;;(set-current-goal 'agent-has-gold)
 ;;;

 ;;;(proof-automath)
 ;;(add-sigma-types-extension 'agent-location (get-lst-proof-of-tdr 'agent-location));;Ok
 ;;(add-sigma-types-extension 'root (list current-agent))
 ;; (get-lst-type (get-lst-of-deepest-context));;Ok
 ;;(eliminated-constraints-double (get-lst-of-all-meronym (get-lst-type (get-lst-of-deepest-context))));;Ok
 ;;(get-lst-of-drt-of-constraints (eliminated-constraints-double (get-lst-of-all-meronym (get-lst-type (get-lst-of-

 )
```

```
;;;; Created on 2008-09-28 12:28:31
;;;Cette fonction permet de calculer le produit cartésien des listes éléments de la liste argument.
;;;
(defun pi-1(arg)
  (first arg)
  )

(defun pi-2(arg)
  (second arg)
  )

(defun del-all-son-of-dtr (drt-name)
  (let ((lst-of-son (get-lst-of-subsuming-element-of-a-type drt-name)))
   (mapcar #'(lambda (x) (if (is-extension x)
                             (progn
                               (initialise-extension-of-sigma x)
                               (del-all-son-of-dtr x)))
                             ) lst-of-son)
   )
  )
;;;;(del-all-son-of-dtr 'agent-location)
;;;(1))
(defun  cartesian-product (list-of-sets)
  (let ((los list-of-sets))
    (if (null los) nil
      (let ((first-set (car los))
            (other-sets (cdr los)))
        (if (null other-sets)
            (mapcar #'list first-set)
          (if (null first-set) nil
            (let ((first-element (car first-set))
                  (other-elements (cdr first-set)))
              (append
               (mapcar #'(lambda (y) (cons first-element y))
                 (cartesian-product other-sets))
               (cartesian-product (cons other-elements other-sets)))))))))))

;;;;(cartesian-product '( (1 2) (a b c) (d) ) )
;;;-------------------------------------------------------------
;;;(2))
(defmacro get-format-name (string-f arg)
  `(format nil (concatenate 'string  ,string-f "~A") ,arg)
  )

;;;Cette fonction permet d'obtenir la liste des arguments d'une sigma type.
;;;(3))
(defun get-lst-types-of-sigma-type(sigma-type-name)
  (let ((arg ))
    (progn
      (setq arg (intern (string-upcase (get-format-name "GET-TYPE-LST-" sigma-type-name))))
    (eval (list arg))
    ))
  )

;;;;(get-lst-types 'agent-location)

;;;-------------------------------------------------------------
;;;(4))
(defun get-extension-lst-of-atomic-types (type-name)
  (gethash type-name agent-kb)
  )
;;;-------------------------------------------------------------
;;;(5))
;;;Les types arguments d'un sigma type sont des types atomiques ou des sigmas.
(defun get-extension-of-types-lst (lst-type-name)
   (mapcar #'(lambda (y)
                          (if (typep y 'sigma-types)
                               (get-extension-of-sigma-types y)
                              (get-extension-lst-of-atomic-types y))) lst-type-name)
  )

;;;(6))
(defmacro get-proof-of-tdr (sigma-type-name lst-of-extension);;Ok
  `(mapcar #'(lambda (x) (apply ,sigma-type-name x)) ,lst-of-extension)
  )
;;;;(get-proof-of-tdr plus1 '(1 2 3))
;;;(7)
(defun remove-nil (lst)
  (delete nil lst)
  )

;;;(8)
(defun get-lst-proof-of-tdr (sigma-type-name);;Ok
    (let* ((lst-of-args (get-lst-types-of-sigma-type sigma-type-name))
           (lst-of-extension (get-extension-of-types-lst lst-of-args))
           (lst-of-cartesion-product-of-extension)
           (lst-of-proof)
```

```
88              (sigma-name sigma-type-name)
               )
90      ;;;Vérifier qu'il existe bien au moins une valeur pour chaque type.
        (if (not (member nil lst-of-extension))
92          (progn
              (setf lst-of-cartesion-product-of-extension (cartesian-product lst-of-extension))
94            (setf lst-of-proof (get-proof-of-tdr sigma-name lst-of-cartesion-product-of-extension))
              (setf lst-of-proof (remove-nil lst-of-proof))
96            )
             )
98      lst-of-proof
    ))
100


102


104
    ;;;Example :
106 ;;(1) agent-location
    ;;(get-lst-types-of-sigma-type 'agent-location);;Ok
108 ;;(setf lst-of-extension  (get-extension-of-types-lst (get-lst-types-of-sigma-type 'agent-location)));;Ok
    ;;(setf lst-of-cartesion-product-of-extension (cartesian-product lst-of-extension));;Ok
110 ;;(get-proof-of-tdr 'agent-location lst-of-cartesion-product-of-extension);;Ok
    ;;(agent-location (first lst-of-cartesion-product-of-extension))
112 ;;(agent-direction (first (first lst-of-cartesion-product-of-extension) ))
    ;;;;;;(get-lst-proof-of-tdr 'agent-location);;Ok
114 ;;(2) ~agent-has-gold
    ;;;;;;(add-sigma-types-extension 'agent-location (get-lst-proof-of-tdr 'agent-location));;Ok
116 ;;(get-lst-types-of-sigma-type '~agent-has-gold);;Ok
    ;;(get-extension-of-sigma-types 'agent-location);;Ok
118 ;;(get-extension-of-types-lst '(agent-location));;Ok
    ;;(setf lst-of-extension  (get-extension-of-types-lst (get-lst-types-of-sigma-type '~agent-has-gold)));;;Ok
120 ;;(setf lst-of-cartesion-product-of-extension (cartesian-product lst-of-extension));;Ok
    ;;(get-proof-of-tdr '~agent-has-gold lst-of-cartesion-product-of-extension)
122
    ;;(has-gold-p (first (first (first lst-of-extension))));;Ok
124 ;;(~has-gold-p (first (first (first lst-of-extension))));;Ok
    ;;;;;;(get-lst-proof-of-tdr '~agent-has-gold);;Ok
126 ;;(3) smelly-square

128 ;;;(add-sigma-types-extension '~agent-has-gold (get-lst-proof-of-tdr '~agent-has-gold));;
    ;;;;;;(get-lst-types-of-sigma-type 'smelly-square);;Ok
130 ;;;;;;(get-extension-of-types-lst '(smelly-square));;Ok
    ;;;;;;(set-agent-position current-agent (make-instance 'position :smelly t))
132 ;;;(get-lst-proof-of-tdr 'smelly-square)

134 ;;(4)next-position-is-a-valid-square
    ;;(get-lst-proof-of-tdr 'next-position-is-a-valid-square)
136

138   (defun get-valid-level (lst)
        (let ((level 0)
140             (current-level))
          ;;;Parcourir les éléments pour décider de la profondeur.
142       (progn
          (loop for x in lst do
144             (progn
                (setq current-level  (first x))
146             (if (< level current-level)
                    (setq level  current-level)
148                 )))
           level)
150       ))

152   (defun get-level-of-a-type(sigma-type-name)
        (let ((arg ))
154     (progn
          (setq arg (intern (string-upcase (get-format-name "GET-LEVEL-" sigma-type-name))))
156     (eval (list arg))
        ))
158   )

160   (defun get-lst-of-subsuming-element-of-a-type(sigma-type-name)
        (let ((arg ))
162     (progn
          (setq arg (intern (string-upcase (get-format-name "GET-SUBSUMING-" sigma-type-name))))
164     (eval (list arg))
        ))
166   )

168   (defun get-lst-of-meronym-element-of-a-type(sigma-type-name)
        (let ((arg ))
170     (progn
```

```
      (setq arg (intern (string-upcase (get-format-name "GET-MERONYM-" sigma-type-name))))
172   (eval (list arg))
      ))
174  )

176 (defun get-lst-of-drt-of-constraint(constraint-name)
      (let ((arg ))
178   (progn
        (setq arg (intern (string-upcase (get-format-name "GET-TYPE-LST-" constraint-name))))
      (eval (list arg))
      ))
182  )

184 ;;(defun execute-action (type-name)
    ;;   (let ((arg ))
186 ;;   (progn
    ;;     (setq arg (intern (string-upcase (get-format-name "EXECUTE-ACTION-" type-name))))
188 ;;   (eval (list arg))
    ;;   ))
190 ;; )
    (defun execute-action (type-name)
192   (let* ((lst-type-name-extension (get-extension-of-sigma-types type-name))
             (sigma-action-type-name (gethash type-name agent-actions-kb)))
194        (mapcar #'(lambda (x)
                    (funcall sigma-action-type-name x)) lst-type-name-extension)
196   ))
    ;;;(execute-action 'next-position-is-a-valid-square)
198 ;;;(get-lst-of-drt-of-constraint dr-01)
    (defun get-lst-of-drt-of-constraints (lst)
200   (mapcar #'(lambda (y)  (list y (get-lst-of-drt-of-constraint y))) lst)
    )
202

204 (defun member-dtr-type (lst value)
      (let ((bValue nil))
206     (mapcar #'(lambda (y)  (if (member value y)
                                 (setf bValue t))) lst)
208   bValue
      )
210   )

212 (defun eliminated-constraints-double (lst)
      (let ((lst-of-constraints))
214    (mapcar #'(lambda (x)  (mapcar #'(lambda (y)  (pushnew y lst-of-constraints)) x))  lst)
      lst-of-constraints)
216   )

218 (defun get-lst-of-drt-expression-of-constraints (lst-of-constraints)
      (let ((lst-details-of-constraints))
220     (mapcar #'(lambda (x)  (pushnew (list x (get-lst-of-drt-of-constraint x)) lst-details-of-constraints) ) lst-of-constrai
      lst-details-of-constraints
222   ))

224 (defun eliminated-no-valid-constraints (lst-of-constraint lst-of-valid-drt)
      (let ((lst-of-valid-constraints)
226         )
      (mapcar #'(lambda (x)  (if (eval (cons 'and (mapcar #'(lambda (y) (member-dtr-type lst-of-valid-drt y)) (second x))))
228                              (pushnew x lst-of-valid-constraints)))  lst-of-constraint)
            lst-of-valid-constraints
230   )
    )
232
    ;;;(setf te1 '((type1 re1) (type2 re2) (type3 re3)))
234 ;;;(setf te2 '((dr1 (type1 type2)) (dr2 (type4 type2)) (dr3 (type5 type6))))
    ;;;(eliminated-no-valid-constraints te2 te1)
236
    (defun get-lst-of-constraints-with-the-biggest-cardinal (lst)
238   (let ((lst-of-valid-constraints)
          (cardinal 0)
240       )
      (progn
242     (mapcar #'(lambda (x) (let ((lst-length (list-length (second x)) ))
                              (if (< cardinal lst-length )
244                               (setf cardinal lst-length)))) lst)
      (mapcar #'(lambda (x) (let ((lst-length (list-length (second x)) ))
246                             (if (= cardinal lst-length)
                                    (pushnew x lst-of-valid-constraints)
248                                 ))) lst)
              lst-of-valid-constraints
250           )))

252 ;;;(setf te3 '((dr1 (type1 type2)) (dr2 (type4 type2 type5)) (dr3 (type5 type6))))
    ;;;(get-lst-of-constraints-with-the-biggest-cardinal  te3)
254
    (defun get-lst-type(lst)
256   (mapcar #'(lambda (x)  (first x))  lst )
    )
258
```

```
(defun get-lst-of-all-meronym (lst)
  (mapcar #'(lambda (x) (get-lst-of-meronym-element-of-a-type x)) lst)
  )


;;;; Created on 2008-10-30 17:25:35
(defun test-conditions()
  (get-lst-proof-of-tdr 'ded-01)
  (execute-action-ded-01)
  )

(defun test-compounds-relations()
  (get-proofs-of-current-compound-dtr)
  )

(defun test-dtr()
  (progn
    (get-proofs-of-current-situation)
    (set-lst-of-the-deepest-dtr))
  )
```

C

Code Lisp du Wumpus

```
;;;; Created on 2008-10-30 22:32:01
(setf stop-value nil)

(setq agent-kb (make-hash-table))
(setq agent-actions-kb (make-hash-table))
(setq agent-compound-types-kb (make-hash-table))

(setf (gethash 'pr1 agent-compound-types-kb)  'next-position-is-a-valid-square)
(setf (gethash 'pr2 agent-compound-types-kb)  'gold-square)
(setf (gethash 'pr3 agent-compound-types-kb)  'next-position-is-a-valid-square)
(setf (gethash 'pr4 agent-compound-types-kb)  'gold-square)
(setf (gethash 'pr5 agent-compound-types-kb)  'next-position-is-a-valid-square)
(setf (gethash 'pr6 agent-compound-types-kb)  'gold-square)

(deftype compound-types ()  '(satisfies compound-types-p))
(defun compound-types-p (type-name)
  (gethash type-name agent-compound-types-kb)
  )


(setf current-agent (make-instance 'agent-wumpus :positionG position11
                                   :direction north))

(setf (gethash 'agent-wumpus agent-kb)  (list current-agent))

(setf (gethash 'direction agent-kb)  (list  ()))

(setf (gethash 'position agent-kb)  (list position11))

(setf (gethash 'has-gold agent-kb)  (list  ()))

(setf (gethash 'position-wumpus agent-kb)  (list  ()))

(setf (gethash 'position-gold agent-kb)  (list  ()))

(setf (gethash 'adjacent-north agent-kb) (list ()))
(setf (gethash 'adjacent-east agent-kb) (list ()))
(setf (gethash 'adjacent-south agent-kb) (list ()))
(setf (gethash 'adjacent-west agent-kb) (list ()))
(setf (gethash 'deductions-lst agent-kb) (list 'ded-01))

(setf (gethash '~agent-has-gold agent-actions-kb) '~agent-has-gold-change-direction)
(setf (gethash 'smelly-square agent-actions-kb) 'smelly-square-change-direction)
(setf (gethash 'breeze-square agent-actions-kb) 'breezy-square-change-direction)
(setf (gethash 'next-position-is-a-valid-square agent-actions-kb) 'next-position-is-a-valid-square-go-forward)
(setf (gethash 'identify-dead-wumpus-square agent-actions-kb) 'identify-dead-wumpus-square-go-forward)
(setf (gethash 'identify-wumpus-square agent-actions-kb) 'identify-wumpus-square-kill-the-wumpus)
(setf (gethash 'gold-square agent-actions-kb) 'gold-square-agent-keep-gold)
(setf (gethash 'agent-has-gold agent-actions-kb) 'agent-has-gold-go-home)

(setf (gethash 'pr1 agent-actions-kb) 'next-position-is-a-valid-square-go-forward)
(setf (gethash 'pr2 agent-actions-kb) 'gold-square-agent-keep-gold)
(setf (gethash 'pr3 agent-actions-kb) 'next-position-is-a-valid-square-go-forward)
(setf (gethash 'pr4 agent-actions-kb) 'gold-square-agent-keep-gold)
(setf (gethash 'pr5 agent-actions-kb) 'next-position-is-a-valid-square-go-forward)
(setf (gethash 'pr6 agent-actions-kb) 'gold-square-agent-keep-gold)


(setf start-position position11)

;;(add-sigma-types-extension 'agent-location (get-lst-proof-of-tdr 'agent-location));;Ok


;;;; Created on 2008-09-28 16:18:15

```

```
  (defclass position ()
4  ((coordinateX :initarg :coordinateX  :accessor coordinateX )
   (coordinateY :initarg :coordinateY  :accessor coordinateY )
6   (valid :initform nil :initarg :valid  :accessor valid )
   (smelly :initform nil :initarg :smelly  :accessor smelly )
8   (breezy :initform nil :initarg :breezy  :accessor breezy )
   (pit :initform nil :initarg :pit  :accessor pit )
10  (visited :initform nil :initarg :visited  :accessor visited )
   (gold :initform nil :initarg :gold  :accessor gold )
12  )
  )
14
 (defclass wall (position)
16  ((valid :initform nil :initarg :valid  :accessor valid ))
  )
18
 (defclass direction ()
20  ()
  )
22
 (defclass east (direction)
24  ()
  )
26 (defclass west (direction)
  ()
28  )
 (defclass north (direction)
30  ()
  )
32 (defclass south (direction)
  ()
34  )

36 (defclass wumpus ()
    ((positionW :initform nil :initarg :positionW  :accessor positionW )
38    (alive  :initform t :initarg :alive  :accessor alive ))
  )
40
 (defclass agent ()
42  ()
  )
44
 (defclass gold ()
46  ()
  )
48
(defclass agent-wumpus (agent)
50  ((positionG :initform 0 :initarg :positionG :accessor positionG)
   (direction :initform nil :initarg :direction :accessor direction)
52  )
  )
54 ;;;;(setq agent1  (make-instance 'agent-wumpus :positionG (make-instance 'position)
;;;;                                              :direction (make-instance 'direction)))
```

```
1 ;;;; Created on 2008-11-01 07:46:12

3 (setq situation (make-hash-table))

5 (setq wall (make-instance 'wall))

7 (setq wumpus1 (make-instance 'wumpus))

9 (setq gold1 (make-instance 'gold))

11 (setq east (make-instance 'east))
  (setq west (make-instance 'west))
13 (setq north (make-instance 'north))
  (setq south (make-instance 'south))
15
  (setq position11 (make-instance 'position :coordinateX 1 :coordinateY 1 :valid t))
17 (setq position12 (make-instance 'position :coordinateX 1 :coordinateY 2 :smelly t))
  (setq position13 (make-instance 'position :coordinateX 1 :coordinateY 3 :pit t))
19 (setq position14 (make-instance 'position :coordinateX 1 :coordinateY 4 :smelly t))
  (setq position15 (make-instance 'position :coordinateX 1 :coordinateY 5))
21
  (setq position21 (make-instance 'position :coordinateX 2 :coordinateY 1 :breezy t))
23 (setq position22 (make-instance 'position :coordinateX 2 :coordinateY 2 :valid t))
  (setq position23 (make-instance 'position :coordinateX 2 :coordinateY 3 :smelly t :breezy t :gold gold1))
25 (setq position24 (make-instance 'position :coordinateX 2 :coordinateY 4  :breezy t))
  (setq position25 (make-instance 'position :coordinateX 2 :coordinateY 5 :valid t))
27
  (setq position31 (make-instance 'position :coordinateX 3 :coordinateY 1 :pit t))
29 (setq position32 (make-instance 'position :coordinateX 3 :coordinateY 2  :breezy t))
  (setq position33 (make-instance 'position :coordinateX 3 :coordinateY 3 :pit t))
31 (setq position34 (make-instance 'position :coordinateX 3 :coordinateY 4  :breezy t))
  (setq position35 (make-instance 'position :coordinateX 3 :coordinateY 5 :valid t))
33
  (setq position41 (make-instance 'position :coordinateX 4 :coordinateY 1 :breezy t))
35 (setq position42 (make-instance 'position :coordinateX 4 :coordinateY 2 :valid t))
  (setq position43 (make-instance 'position :coordinateX 4 :coordinateY 3  :breezy t))
37 (setq position44 (make-instance 'position :coordinateX 4 :coordinateY 4 :pit t))
```

```
(setq position45 (make-instance 'position :coordinateX 4 :coordinateY 5 :valid t))

(setq position51 (make-instance 'position :coordinateX 5 :coordinateY 1 :valid t))
(setq position52 (make-instance 'position :coordinateX 5 :coordinateY 2 :valid t))
(setq position53 (make-instance 'position :coordinateX 5 :coordinateY 3 :valid t))
(setq position54 (make-instance 'position :coordinateX 5 :coordinateY 4 :valid t))
(setq position55 (make-instance 'position :coordinateX 5 :coordinateY 5 :valid t))

(setf (gethash 'adjacent-east situation)   (list
                                            (list position51 wall)
                                            (list position52 wall)
                                            (list position53 wall)
                                            (list position54 wall)
                                            (list position55 wall)
                                            (list position11 position21)
                                            (list position21 position31)
                                            (list position31 position41)
                                            (list position41 position51)
                                            (list position12 position22)
                                            (list position22 position32)
                                            (list position32 position42)
                                            (list position42 position52)
                                            (list position13 position23)
                                            (list position23 position33)
                                            (list position33 position43)
                                            (list position43 position53)
                                            (list position14 position24)
                                            (list position24 position34)
                                            (list position34 position44)
                                            (list position44 position54)
                                            (list position15 position25)
                                            (list position25 position35)
                                            (list position35 position45)
                                            (list position45 position55)
                                            ))

(setf (gethash 'adjacent-north situation)  (list
                                            (list position15 wall)
                                            (list position25 wall)
                                            (list position35 wall)
                                            (list position45 wall)
                                            (list position55 wall)
                                            (list position11 position12)
                                            (list position21 position22)
                                            (list position31 position32)
                                            (list position41 position42)
                                            (list position51 position52)
                                            (list position12 position13)
                                            (list position22 position23)
                                            (list position32 position33)
                                            (list position42 position43)
                                            (list position52 position53)
                                            (list position13 position14)
                                            (list position23 position24)
                                            (list position33 position34)
                                            (list position43 position44)
                                            (list position53 position54)
                                            (list position14 position15)
                                            (list position24 position25)
                                            (list position34 position35)
                                            (list position44 position45)
                                            (list position54 position55)
                                            ))

(setf (gethash 'adjacent-south situation) (list
                                            (list position11 wall)
                                            (list position21 wall)
                                            (list position31 wall)
                                            (list position41 wall)
                                            (list position51 wall)
                                            (list position12 position11)
                                            (list position22 position21)
                                            (list position32 position31)
                                            (list position42 position41)
                                            (list position52 position51)
                                            (list position13 position12)
                                            (list position23 position22)
                                            (list position33 position32)
                                            (list position43 position42)
                                            (list position53 position52)
                                            (list position14 position13)
                                            (list position24 position23)
                                            (list position34 position33)
                                            (list position44 position43)
                                            (list position54 position53)
                                            (list position15 position14)
                                            (list position25 position24)
                                            (list position35 position34)
                                            (list position45 position44)
                                            (list position55 position54)
```

```
                                          ))
131
(setf (gethash 'adjacent-west situation) (list
133                                         (list position11 wall)
                                            (list position12 wall)
135                                         (list position13 wall)
                                            (list position14 wall)
137                                         (list position15 wall)
                                            (list position21 position11)
139                                         (list position31 position21)
                                            (list position41 position31)
141                                         (list position51 position41)
                                            (list position22 position12)
143                                         (list position32 position22)
                                            (list position42 position32)
145                                         (list position52 position42)
                                            (list position23 position13)
147                                         (list position33 position23)
                                            (list position43 position33)
149                                         (list position53 position43)
                                            (list position24 position14)
151                                         (list position34 position24)
                                            (list position44 position34)
153                                         (list position54 position44)
                                            (list position25 position15)
155                                         (list position35 position25)
                                            (list position45 position35)
157                                         (list position55 position45)
                                            ))
159

161 (setf (gethash 'direction situation)  (list  east
                                            west
163                                         north
                                            south))
165
(setf (gethash 'position situation)  (list position11 position12 position13 position14 position15
167                                         position21 position22 position23 position24 position25
                                            position31 position32 position33 position34 position35
169                                         position41 position42 position43 position44 position45
                                            position51 position52 position53 position54 position55))
171

173
(setf (gethash 'has-gold situation)  (list '()))
175
(setf (gethash 'position-wumpus situation)  (list  position13 wumpus1))
177
(setf (gethash 'position-gold situation)  (list  position23 gold1))


 1 ;;;; Created on 2008-10-30 17:27:33
(defun agent-position(agent-wump)
 3   (positionG agent-wump)
   )
 5 ;;agent-direction : (Pi a:agent-wumpus.direction(a)))
(defun agent-direction(agent-wump)
 7   (direction agent-wump)
   )
 9
(defun set-agent-position (agent-wump positionEC)
11   (setf (positionG agent-wump) positionEC)
   )
13
(defun get-inverse-direction (direction)
15    (cond
      ((equal direction north) south)
17    ((equal direction east)  west)
      ((equal direction south) north)
19    ((equal direction west)  east))
   )
21 ;;;(get-inverse-direction 'north)

23 (defun get-next-left-direction (direction)
    (cond
25    ((equal direction north) west)
      ((equal direction west)  south)
27    ((equal direction south) east)
      ((equal direction east)  north))
29 )
 ;;;(get-next-left-direction 'north)
31 (defun next-position-kb (position direction)
   (cond
33    ((equal direction north) (next-north-position-kb position))
      ((equal direction east)  (next-east-position-kb position))
35    ((equal direction south) (next-south-position-kb position))
      ((equal direction west)  (next-west-position-kb  position)))
37 )

39
(defun next-position (position direction)
```

```
41   (cond
       ((equal direction north) (next-north-position position))
43     ((equal direction east)  (next-east-position position))
       ((equal direction south) (next-south-position position))
45     ((equal direction west)  (next-west-position  position)))
     )
47  ;;;Example
;;(next-position 'position12 'east)
49   (defun next-east-position (position)
       (let ((lst (gethash 'adjacent-east situation))
51           (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
53                               (setf next-position (second x) )
                                 )) lst)
55     next-position
       )
57     )

59   (defun next-east-position-kb (position)
       (let ((lst (gethash 'adjacent-east agent-kb))
61           (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
63                               (setf next-position (second x) )
                                 )) lst)
65     next-position
       )
67     )

69   (defun next-south-position (position)
       (let ((lst (gethash 'adjacent-south situation))
71           (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
73                               (setf next-position (second x) )
                                 )) lst)
75     next-position
       )
77     )

79  (defun next-south-position-kb (position)
       (let ((lst (gethash 'adjacent-south agent-kb))
81           (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
83                               (setf next-position (second x) )
                                 )) lst)
85     next-position
       )
87     )

89   (defun next-west-position (position)
       (let ((lst (gethash 'adjacent-west situation))
91           (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
93                               (setf next-position (second x) )
                                 )) lst)
95     next-position
       )
97     )

99   (defun next-west-position-kb (position)
       (let ((lst (gethash 'adjacent-west agent-kb))
101          (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
103                              (setf next-position (second x) )
                                 )) lst)
105    next-position
       )
107    )

109  (defun next-north-position (position)
       (let ((lst (gethash 'adjacent-north situation))
111          (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
113                              (setf next-position (second x) )
                                 )) lst)
115    next-position
       )
117    )

119  (defun next-north-position-kb (position)
       (let ((lst (gethash 'adjacent-north agent-kb))
121          (next-position nil))
       (mapcar #'(lambda (x) (if (equal (first x) position)
123                              (setf next-position (second x) )
                                 )) lst)
125    next-position
       )
127    )

129 (defun get-the-wumpus (position)
     (let ((lst (gethash 'adjacent-east situation))
131          (wumpus-pos nil))
```

```
      (mapcar #'(lambda (x) (if (equal (first x) position)
                                (setf wumpus-pos (second x) )
                                )) lst)
    wumpus-pos
     )
    )

(defun is-direction-known(direction position)
  (let ((lst-direction)
          (positionS)
        )
    (progn
      (cond
        ((equal direction north) (setf lst-direction (gethash 'adjacent-north agent-kb) ))
        ((equal direction east)   (setf lst-direction (gethash 'adjacent-east agent-kb)) )
        ((equal direction south) (setf lst-direction (gethash 'adjacent-south agent-kb)) )
        ((equal direction west)   (setf lst-direction (gethash 'adjacent-west agent-kb)) ))
      ;;(print lst-direction)
      (mapcar #'(lambda (x) (if (equal (first x) position)
                                (setf positionS (second x) )
                                )) lst-direction)
    positionS )
  )
  )


(defun is-start-position (current-position)
  (equal start-position current-position)
  )

(defun set-position-valid(position)
  (setf  (valid position) t)
  )

(defun choose-direction (lst-direction position)
  (let ((choice-direction nil)
          (next-position nil))
    (progn
      (mapcar #'(lambda (x) ( progn
                              (setf  next-position (next-position-kb position x))
                              (if (equal next-position nil)
                                  (if (equal choice-direction nil)
                                      (setf choice-direction x)
                                      )))
                  ) lst-direction )
    choice-direction)
  ))


  (defun inference-forward-position (position direction)
    (let ((next-pos (next-position position direction))
             (inverse-direction  (get-inverse-direction direction))
          )
      (progn
        (print "inference-forward-position")
        (setf (visited next-pos) t)
        (pushnew  next-pos (gethash 'position agent-kb))
        (cond
          ((equal direction north)  (pushnew (list position next-pos) (gethash 'adjacent-north agent-kb)))
          ((equal direction east)    (pushnew (list position next-pos) (gethash 'adjacent-east agent-kb)))
          ((equal direction south) (pushnew (list position next-pos) (gethash 'adjacent-south agent-kb) ))
          ((equal direction west)   (pushnew (list position next-pos) (gethash 'adjacent-west agent-kb) )))
        (cond
          ((equal inverse-direction north)  (pushnew (list next-pos position) (gethash 'adjacent-north agent-kb) ))
          ((equal inverse-direction east)    (pushnew (list next-pos position) (gethash 'adjacent-east agent-kb) ))
          ((equal inverse-direction south) (pushnew (list next-pos position) (gethash 'adjacent-south agent-kb) ))
          ((equal inverse-direction west)   (pushnew (list next-pos position) (gethash 'adjacent-west agent-kb) )))
        next-pos
        )
      ))
```

```
;;;; Created on 2008-10-26 17:45:59
;;agent-position : (Pi a:agent-wumpus.position(a)))
(defun agent-position(agent-wump)
  (positionG agent-wump)
  )
;;agent-direction : (Pi a:agent-wumpus.direction(a)))
(defun agent-direction(agent-wump)
  (direction agent-wump)
  )
;;has-goldp : (Pi a:agent-wumpus.prop(a)))
(defun has-goldp(agent-wump)
  (let((lst (gethash 'has-gold kb-agent))
         (bValue nil))
    (mapcar #'(lambda (x) (if (equal (first x) agent-wump)
                              (setf bValue t)
                              )) lst)
    bValue
    )
```

```
  )
;;~has-goldp : (Pi a:agent-wumpus.prop(a)))
(defun ~has-goldp(agent-wump)
         (equal (has-goldp agent-wump) nil)
  )

(defun set-agent-position (agent-wump positionEC)
  (setf (positionG agent-wump) positionEC)
  )

(defun is-smelly-positionp (positionEC)
  (smelly positionEC)
  )


;;;; Created on 2008-10-30 17:27:18
;;has-goldp : (Pi a:agent-wumpus.prop(a)))
(defun has-gold-p(agent-wump)
  (let((lst (gethash 'has-gold agent-kb))
         (bValue nil))
    (mapcar #'(lambda (x) (if (equal (first x) agent-wump)
                              (setf bValue t)
                              )) lst)
    bValue
    )
  )
;;~has-goldp : (Pi a:agent-wumpus.prop(a)))
(defun ~has-gold-p(agent-wump)
         (equal (has-gold-p agent-wump) nil)
  )

(defun is-smelly-position-p (positionEC)
  (smelly positionEC)
  )

(defun is-valid-position-p (positionEC)
  (if (equal positionEC nil)
      nil
      (valid positionEC))
  )

(defun is-different-position-p (position1 position2)
 (not  (equal position1 position2))
  )

(defun is-adjacent-position-p (position1 position2)
  (or (equal (adjacent-north position1) position2)
       (equal (adjacent-east position1) position2)
       (equal (adjacent-south position1) position2)
       (equal (adjacent-west position1) position2))
  )

(defun is-wumpus-dead-p (wumpusP)
  (not  (alive wumpusP))
  )

(defun is-wumpus-located-in-p (position)
  (let ((lst (gethash 'position-wumpus agent-kb))
          (bValue nil))
    (mapcar #'(lambda (x) (if (equal (first x) position)
                              (setf bValue t)
                              )) lst)
    bValue
    )
  )

(defun is-perceive-glitter-p (position)
  ( gold position)
  )

(defun is-breezy-position-p (positionEC)
  (breezy positionEC)
  )

(defun is-member-p (el1 lst)
  (let ((bValue nil))
   (mapcar #'(lambda (x) (if (equal x el1)
                             (setf bValue t)
                             )) lst)
    bValue
  ))

(defun is-adjacent-p (position1 position2)
  (let ((lst1 (gethash 'adjacent-east agent-kb))
          (lst2 (gethash 'adjacent-west agent-kb))
          (lst3 (gethash 'adjacent-north agent-kb))
          (lst4 (gethash 'adjacent-south agent-kb)))
  (or (is-member-p (list position1 position2) lst1)
        (is-member-p (list position1 position2) lst2)
        (is-member-p (list position1 position2) lst3)
        (is-member-p (list position1 position2) lst4))
```

```
  ))
;;;Example :
;;(is-adjacent-p position11 position12)

(defun is-different-p (position1 position2)
  (not(equal position1 position2))
  )

(defun is-possible-direction(position direction)
  (let ((next-pos (next-position-kb position direction)))
    (and (or (is-valid-position-p next-pos) (is-valid-position-p position)) (not(equal next-pos wall)))
    )
  )
```

```
;;;; Created on 2008-10-26 19:12:34

(add-sigma-types-extension 'agent-location '())
(add-sigma-types-extension 'agent-has-gold '())
(add-sigma-types-extension '~agent-has-gold '())
(add-sigma-types-extension 'breeze-square '())
(add-sigma-types-extension 'gold-square '())
(add-sigma-types-extension 'identify-dead-wumpus-square '())
(add-sigma-types-extension 'identify-wumpus-square '())
(add-sigma-types-extension 'next-position-is-a-valid-square '())
(add-sigma-types-extension 'position-in-valid-square '())
(add-sigma-types-extension 'smelly-square '())

(add-sigma-types-extension 'root (list current-agent))
```

```
;;;; Created on 2008-11-23 13:05:28
;;;; Created on 2008-10-09 14:41:25
;;;; ~agent-has-gold

(let ((name '~agent-has-gold-change-direction)
        (lst-types '(~agent-has-gold))
        (subsumed '())
        (subsuming '())
        (meronym '())
        (holonym '())
        (current-value '())
        (level 2)
        (constructor  '(lambda (x)
                         (let*((a x)
                                 (b (change-direction (pi-1 (pi-1 a))
                                                      (pi-1(pi-2 (pi-1 a)))
                                                      (pi-2(pi-2(pi-1  a))))))
                           (progn
                           (if  (not(equal b nil))
                              (list a b)
                               nil
                               ))
                           )
                         ))
  (defun get-level-~agent-has-gold-change-direction()
    level
    )
  (defun initialise-~agent-has-gold-change-direction()
    (setf current-value '())
    )
  (defun get-subsumed-~agent-has-gold-change-direction ()
    subsumed
    )
  (defun get-subsuming-~agent-has-gold-change-direction  ()
    subsuming
    )
  (defun get-meronym-~agent-has-gold-change-direction  ()
    meronym
    )
  (defun get-holonym-~agent-has-gold-change-direction  ()
    holonym
    )
  (defun proof-~agent-has-gold-change-direction ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-~agent-has-gold-change-direction ()
    lst-types
    )
  (defun  ~agent-has-gold-change-direction (~agent-has-gold)
    (funcall (eval constructor ) ~agent-has-gold)))
```

```
;;;; Created on 2008-11-23 13:05:28
;;;; Created on 2008-10-09 14:41:25
;;;; ~agent-has-gold

(let ((name 'smelly-square-change-direction)
        (lst-types '(smelly-square))
        (subsumed '())
```

```
        (subsuming '())
        (meronym '())
        (holonym '())
        (current-value '())
        (level 2)
        (constructor  '(lambda (x)
                         (let*((a x)
                                 (b (change-direction (pi-1 (pi-1(pi-1 a)))
                                                      (pi-1(pi-2 (pi-1(pi-1  a))))
                                                      (pi-2(pi-2 (pi-1(pi-1  a))))
                                                      )))
                           (progn
                           (if  (not(equal b nil))
                              (list a b)
                               nil
                               ))
                           )
                          ))
  (defun get-level-smelly-square-change-direction()
    level
    )
  (defun initialise-smelly-square-change-direction()
    (setf current-value '())
    )
  (defun get-subsumed-smelly-square-change-direction ()
    subsumed
    )
  (defun get-subsuming-smelly-square-change-direction  ()
    subsuming
    )
  (defun get-meronym-smelly-square-change-direction  ()
    meronym
    )
  (defun get-holonym-smelly-square-change-direction  ()
    holonym
    )
  (defun proof-smelly-square-change-direction ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-smelly-square-change-direction ()
    lst-types
    )
  (defun  smelly-square-change-direction (smelly-square)
    (funcall (eval constructor ) smelly-square)))

;;;; Created on 2008-11-23 13:05:28
;;;; Created on 2008-10-09 14:41:25
;;;; ~agent-has-gold

(let ((name 'breezy-square-change-direction)
        (lst-types '(breezy-square))
        (subsumed '())
        (subsuming '())
        (meronym '())
        (holonym '())
        (current-value '())
        (level 2)
        (constructor  '(lambda (x)
                         (let*((a x)
                                 (b (change-direction (pi-1 (pi-1(pi-1 a)))
                                                      (pi-1(pi-2 (pi-1(pi-1  a))))
                                                      (pi-2(pi-2 (pi-1(pi-1  a))))
                                                      )))
                           (progn
                           (if  (not(equal b nil))
                              (list a b)
                               nil
                               ))
                           )
                          ))
  (defun get-level-breezy-square-change-direction()
    level
    )
  (defun initialise-breezy-square-change-direction()
    (setf current-value '())
    )
  (defun get-subsumed-breezy-square-change-direction ()
    subsumed
    )
  (defun get-subsuming-breezy-square-change-direction  ()
    subsuming
    )
  (defun get-meronym-breezy-square-change-direction  ()
    meronym
    )
  (defun get-holonym-breezy-square-change-direction  ()
    holonym
    )
  (defun proof-breezy-square-change-direction ()
```

```
46      ;;;Call a generic function.
        )
48    (defun get-type-lst-breezy-square-change-direction ()
        lst-types
50      )
      (defun  breezy-square-change-direction (breezy-square)
52      (funcall (eval constructor ) breezy-square)))


    ;;;; Created on 2008-11-23 13:05:28
2   ;;;; Created on 2008-10-09 14:41:25
    ;;;; ~agent-has-gold
4
    (let ((name 'next-position-is-a-valid-square-go-forward)
6           (lst-types '(next-position-is-a-valid-square))
            (subsumed '())
8           (subsuming '())
            (meronym '())
10          (holonym '())
            (current-value '())
12          (level 2)
            (constructor  '(lambda (x)
14                           (let*((a x)
                                   (b (go-forward (pi-1 (pi-1(pi-1 a)))
16                                                (pi-1(pi-2 (pi-1(pi-1  a))))
                                                  (pi-2(pi-2 (pi-1(pi-1  a))))
18                                                        )))
                               (progn
20                             (if  (not(equal b nil))
                                  (list a b)
22                                nil
                                  ))
24                             )
                              )
26                         ))
      (defun get-level-next-position-is-a-valid-square-go-forward()
28      level
        )
30    (defun initialise-next-position-is-a-valid-square-go-forward()
        (setf current-value '())
32      )
      (defun get-subsumed-next-position-is-a-valid-square-go-forward ()
34      subsumed
        )
36    (defun get-subsuming-next-position-is-a-valid-square-go-forward  ()
        subsuming
38      )
      (defun get-meronym-next-position-is-a-valid-square-go-forward  ()
40      meronym
        )
42    (defun get-holonym-next-position-is-a-valid-square-go-forward  ()
        holonym
44      )
      (defun proof-next-position-is-a-valid-square-go-forward ()
46      ;;;Call a generic function.
        )
48    (defun get-type-lst-next-position-is-a-valid-square-go-forward ()
        lst-types
50      )
      (defun  next-position-is-a-valid-square-go-forward (next-position-is-a-valid-square)
52      (funcall (eval constructor ) next-position-is-a-valid-square)))


    ;;;; Created on 2008-11-23 13:05:28
2   ;;;; Created on 2008-10-09 14:41:25
    ;;;; ~agent-has-gold
4
    (let ((name 'identify-dead-wumpus-square-go-forward)
6           (lst-types '(identify-dead-wumpus-square))
            (subsumed '())
8           (subsuming '())
            (meronym '())
10          (holonym '())
            (current-value '())
12          (level 2)
            (constructor  '(lambda (x)
14                           (let*((a x)
                                   (b (go-forward (pi-1 (pi-1 (pi-1(pi-1 a))))
16                                                (pi-1(pi-2 (pi-1 (pi-1(pi-1  a)))))
                                                  (pi-2(pi-2 (pi-1 (pi-1(pi-1  a)))))
18                                                        )))
                               (progn
20                             (if  (not(equal b nil))
                                  (list a b)
22                                nil
                                  ))
24                             )
                              )
26                         ))
      (defun get-level-identify-dead-wumpus-square-go-forward()
28      level
```

```
    )
30  (defun initialise-identify-dead-wumpus-square-go-forward()
      (setf current-value '())
32    )
    (defun get-subsumed-identify-dead-wumpus-square-go-forward ()
34    subsumed
      )
36  (defun get-subsuming-identify-dead-wumpus-square-go-forward  ()
      subsuming
38    )
    (defun get-meronym-identify-dead-wumpus-square-go-forward  ()
40    meronym
      )
42  (defun get-holonym-identify-dead-wumpus-square-go-forward  ()
      holonym
44    )
    (defun proof-identify-dead-wumpus-square-go-forward ()
46    ;;;Call a generic function.
      )
48  (defun get-type-lst-identify-dead-wumpus-square-go-forward ()
      lst-types
50    )
    (defun  identify-dead-wumpus-square-go-forward (identify-dead-wumpus-square)
52    (funcall (eval constructor ) identify-dead-wumpus-square)))


;;;; Created on 2008-11-23 13:05:28
2 ;;;; Created on 2008-10-09 14:41:25
  ;;;; ~agent-has-gold
4
  (let ((name 'identify-wumpus-square-kill-wumpus)
6         (lst-types '(identify-wumpus-square))
          (subsumed '())
8         (subsuming '())
          (meronym '())
10        (holonym '())
          (current-value '())
12        (level 2)
          (constructor  '(lambda (x)
14                         (let*((a x)
                                 (b (killed-the-wumpus (pi-1 (pi-1 (pi-1(pi-1 a))))
16                                                       (pi-1(pi-2 (pi-1 (pi-1(pi-1  a)))))
                                                      )))
18                           (progn
                             (if  (not(equal b nil))
20                               (list a b)
                                  nil
22                                ))
                             )
24                          )
                       ))
26  (defun get-level-identify-wumpus-square-kill-wumpus()
      level
28    )
    (defun initialise-identify-wumpus-square-kill-wumpus()
30    (setf current-value '())
      )
32  (defun get-subsumed-identify-wumpus-square-kill-wumpus ()
      subsumed
34    )
    (defun get-subsuming-identify-wumpus-square-kill-wumpus  ()
36    subsuming
      )
38  (defun get-meronym-identify-wumpus-square-kill-wumpus  ()
      meronym
40    )
    (defun get-holonym-identify-wumpus-square-kill-wumpus ()
42    holonym
      )
44  (defun proof-identify-wumpus-square-kill-wumpus ()
      ;;;Call a generic function.
46    )
    (defun get-type-lst-identify-wumpus-square-kill-wumpus ()
48    lst-types
      )
50  (defun  identify-wumpus-square-kill-wumpus (identify-wumpus-square)
      (funcall (eval constructor ) identify-wumpus-square)))


1 ;;;; Created on 2008-11-23 13:05:28
  ;;;; Created on 2008-10-09 14:41:25
3 ;;;; ~agent-has-gold

5 (let ((name 'gold-square-agent-keep-gold)
          (lst-types '(gold-square))
7         (subsumed '())
          (subsuming '())
9         (meronym '())
          (holonym '())
11        (current-value '())
          (level 2)
```

```
13          (constructor   '(lambda (x)
                              (let*((a x)
15                                  (b (keep-gold  (pi-1 (pi-1(pi-1 a))) (pi-2 a)
                                                                          )))
17                              (progn
                                (if  (not(equal b nil))
19                                 (list a b)
                                    nil
21                                  ))
                                )
23                              )
                            ))
25    (defun get-level-gold-square-agent-keep-gold()
        level
27      )
      (defun initialise-gold-square-agent-keep-gold()
29      (setf current-value '())
        )
31    (defun get-subsumed-gold-square-agent-keep-gold ()
        subsumed
33      )
      (defun get-subsuming-gold-square-agent-keep-gold  ()
35      subsuming
        )
37    (defun get-meronym-gold-square-agent-keep-gold  ()
        meronym
39      )
      (defun get-holonym-gold-square-agent-keep-gold ()
41      holonym
        )
43    (defun proof-gold-square-agent-keep-gold ()
        ;;;Call a generic function.
45      )
      (defun get-type-lst-gold-square-agent-keep-gold ()
47      lst-types
        )
49    (defun  gold-square-agent-keep-gold (gold-square)
        (funcall (eval constructor ) gold-square)))


    ;;;; Created on 2008-11-23 13:05:28
2   ;;;; Created on 2008-10-09 14:41:25
    ;;;; ~agent-has-gold
4
    (let ((name 'agent-has-gold-go-home)
6           (lst-types '(agent-has-gold))
            (subsumed '())
8           (subsuming '())
            (meronym '())
10          (holonym '())
            (current-value '())
12          (level 2)
            (constructor   '(lambda (x)
14                            (let*((a x)
                                    (b (go-home
16                                                          )))
                                (progn
18                              (if  (not(equal b nil))
                                   (list a b)
20                                  nil
                                    ))
22                              )
                                )
24                          ))
      (defun get-level-agent-has-gold-go-home()
26      level
        )
28    (defun initialise-agent-has-gold-go-home()
        (setf current-value '())
30      )
      (defun get-subsumed-agent-has-gold-go-home ()
32      subsumed
        )
34    (defun get-subsuming-agent-has-gold-go-home  ()
        subsuming
36      )
      (defun get-meronym-agent-has-gold-go-home  ()
38      meronym
        )
40    (defun get-holonym-agent-has-gold-go-home ()
        holonym
42      )
      (defun proof-agent-has-gold-go-home ()
44      ;;;Call a generic function.
        )
46    (defun get-type-lst-agent-has-gold-go-home ()
        lst-types
48      )
      (defun  agent-has-gold-go-home (agent-has-gold)
50      (funcall (eval constructor ) agent-has-gold)))
```

```
;;;; Created on 2008-10-09 22:00:10
(let ((lst-types '(agent-direction-to-valid-square))
      (constructor  '(lambda (x)
                       (let*((a x)
                             (b (funcall  (agent-move-to) (first a) (first (second a)))))
                         (if (not(equal b nil))
                             (list a b)
                             nil
                             )))))
  (defun proof-move-agent-to-valid-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-move-agent-to-valid-square ()
    lst-types
    )
  (defun  move-agent-to-valid-square (agent-wump)
    (funcall (eval constructor ) agent-wump)))


;;;; Created on 2008-10-09 21:23:32
;;;;move-agent-to-valid-square : (Pi a : agent-to-valid-square.agent-move-to(pi(a) pi(pi'(pi'(pi'(pi'(a)))))))
;;;;agent-keep-gold       : (Pi a : gold-square().agent-keep-gold(pi(a) pi(pi'(a) )))
;;;;agent-change-direction    : (Pi a : smelly-square.change-direction(pi(a) pi(pi'(pi'(a) )))
;;;;kill-the-wumpus       : (Pi a : identify-wumpus-position.kill-wumpus(pi(pi(a))  pi(pi'(pi'(a))))
(let ((return-type 'root))
  (defun go-home()
    (progn
      (print "go-home")
      (set-stop-active-agent)
      (set-current-type-if-low-level  return-type)
      )
    ))

(let ((return-type 'agent-location))
  (defun keep-gold (agent gold)
    (progn
       (print "keep-gold")
      (push (list agent gold) (gethash 'has-gold agent-kb))
      (set-current-type-if-low-level  return-type)
      )
    )
)

;;(let ((return-type 'root))
;;  (defun change-direction (agent position direction)
;;    (let (
;;          (lst-possible '())
;;          (lst-last (list (get-inverse-direction direction)))
;;          (lst-impossible '())
;;          (current-direction direction)
;;          (last-position)
;;          )
;;      (progn
;;         (print "change-direction")
;;         (print current-direction)
;;         (loop for i from 1 to 3 do
;;               (progn
;;                 (setf  current-direction (get-next-left-direction current-direction))
;;                 (setf  last-position (is-direction-known current-direction position))
;;                 (if  (not  (equal last-position nil))
;;                     (if (and  (valid last-position) (not (equal wall last-position)))
;;                         (setf lst-possible (append lst-possible (list current-direction)))
;;                         (setf lst-impossible (append lst-possible (list current-direction)))
;;                         )
;;                     (setf lst-possible (append lst-possible (list current-direction)))
;;                     ))
;;               )
;;         (if (not (equal lst-possible nil))
;;             (progn
;;               (setf (direction agent) (choose-direction lst-possible position))
;;               (if (equal (direction agent) nil)
;;                   (setf (direction agent) (first lst-possible))))
;;               (setf (direction agent) (first lst-last)))
;;         (set-current-type-if-low-level  return-type)
;;         )
;;       )
;;     )
;;   )
(defun change-direction-free (agent position direction)
```

```
  (let (
          (lst-possible nil)
          (current-direction direction)
          (positionEC)
          )
      (progn
        (print "change-direction-free")
        (loop for i from 1 to 3 do
              (progn
                (setf  current-direction (get-next-left-direction current-direction))
                (setf  positionEC (is-direction-known current-direction position))
                (if  (equal positionEC nil)
                     (pushnew current-direction lst-possible))
                ))
         (if (not (equal lst-possible nil))
             (setf (direction agent) (first lst-possible))
             (setf (direction agent) (get-next-left-direction direction)))
        )))

(defun change-direction (agent position direction)
  (let (
          (lst-possible nil)
          (current-direction direction)
          (positionEC)
          )
      (progn
        (print "change-direction")
        (loop for i from 1 to 3 do
              (progn
                (setf  current-direction (get-next-left-direction current-direction))
                (setf  positionEC (is-direction-known current-direction position))
                (if  (not (equal positionEC nil))
                     (if (and (valid positionEC) (not (visited positionEC)))
                     (pushnew current-direction lst-possible))
                     )))
        (print lst-possible)
         (if (not (equal lst-possible nil))
             (setf (direction agent) (first lst-possible))
             (setf (direction agent) (get-next-left-direction direction)))
        )))

;;(change-direction current-agent (positionG current-agent) (direction current-agent))
;;;Example
;;(setf (smelly position12) t)
;;(pushnew (list position11 position12) (gethash 'adjacent-north agent-kb)  )
;;(pushnew (list position11 wall) (gethash 'adjacent-west agent-kb)  )
;;(change-direction current-agent position11 south)
(let ((return-type 'identify-wumpus-square))
  (defun killed-the-wumpus (agent position)
    (let ((next-pos (next-position position direction)))
      (prog
       (print "killed-the-wumpus")
       (get-the-wumpus next-position)
       (set-current-type-if-low-level  return-type)
       )
      ))
)

(let ((return-type 'root))
  (defun go-forward (agent position direction)
    (let ((next-pos (inference-forward-position position direction))
          )
      (progn
        (print "go-forward")
        (if (not (equal next-pos wall))
              (setf (positionG agent) next-pos))
        (set-current-type-if-low-level  return-type)
        )
      )
    )
  )
;;-----------------------------------------------------------------------------------------------------------------------
```

```
;;;; Created on 2008-10-25 12:19:17
(defun set-valid-square (cposition)
  (setf (valid cposition) t)
  )

;;;; Created on 2008-10-18 15:30:36
;;;pr1.
(let ((name 'pr1)
        (lst-types '(next-position-is-a-valid-square breeze-square))
        (subsumed '())
        (subsuming '(pr3))
        (meronym '())
        (holonym 'execute-action-next-position-is-a-valid-square)
        (constructor  '(lambda (x y)
                         (let*((a x)
                               (b y)
```

```
12                              (c (funcall  (part-of) a b)))
                            (if (not(equal c nil))
14                              (list a (list b c))
                                nil
16                              )))))
    (defun get-subsumed-pr1 ()
18    subsumed
      )
20  (defun get-subsuming-pr1  ()
      subsuming
22    )
    (defun get-meronym-pr1  ()
24    meronym
      )
26  (defun get-holonym-pr1  ()
      holonym
28    )
    (defun proof-pr1 ()
30    ;;;Call a generic function.
      )
32  (defun get-type-lst-pr1 ()
      lst-types
34    )
    (defun  pr1 (agent-direction-to-valid-square breeze-square)
36    (funcall (eval constructor ) agent-direction-to-valid-square breeze-square)))


;;;; Created on 2008-10-18 15:30:36
2 ;;;pr1.
  (let ((name 'pr2)
4         (lst-types '(breeze-square gold-square))
          (subsumed '())
6         (subsuming '())
          (meronym '())
8         (holonym nil)
          (constructor  '(lambda (x y)
10                         (let*((a x)
                                 (b y)
12                               (c (funcall  (part-of) a b)))
                             (if (not(equal c nil))
14                               (list a (list b c))
                                 nil
16                               )))))
    (defun get-subsumed-pr2 ()
18    subsumed
      )
20  (defun get-subsuming-pr4  ()
      subsuming
22    )
    (defun get-meronym-pr2  ()
24    meronym
      )
26  (defun get-holonym-pr2  ()
      holonym
28    )
    (defun proof-pr2 ()
30    ;;;Call a generic function.
      )
32  (defun get-type-lst-pr2 ()
      lst-types
34    )
    (defun  pr2 (breeze-square gold-square)
36    (funcall (eval constructor ) breeze-square gold-square)))


;;;; Created on 2008-10-18 15:42:15
2 (let  ((name 'pr3)
          (lst-types '(next-position-is-a-valid-square breeze-square smelly-square))
4         (subsumed '(pr1 pr2))
          (subsuming '(pr4))
6         (meronym '())
          (holonym '(next-position-is-a-valid-square breeze-square smelly-square))
8         (constructor  '(lambda (x y z)
                           (let*((a x)
10                               (b y)
                                 (c z)
12                               (d (funcall  (part-of) a b c)))
                             (if (not(equal d nil))
14                               (list a b c d)
                                 nil
16                               )))))
    (defun get-subsumed-pr3 ()
18    subsumed
      )
20  (defun get-subsuming-pr3  ()
      subsuming
22    )
    (defun get-meronym-pr3  ()
24    meronym
      )
26  (defun get-holonym-pr3  ()
      holonym
```

```
28      )
    (defun proof-pr3 ()
30      ;;;Call a generic function.
      )
32    (defun get-type-lst-pr3 ()
      lst-types
34      )
    (defun  pr3 (agent-direction-to-valid-square breeze-square smelly-square)
36      (funcall (eval constructor ) agent-direction-to-valid-square breeze-square smelly-square)))


  ;;;; Created on 2008-10-18 15:30:36
2 ;;;pr1.
  (let ((name 'pr4)
4         (lst-types '(smelly-square breeze-square gold-square))
          (subsumed '())
6         (subsuming '())
          (meronym '())
8         (holonym nil)
          (constructor  '(lambda (x y z)
10                         (let*((a x)
                                 (b y)
12                               (c z)
                                 (d (funcall  (part-of) a b c)))
14                           (if (not(equal d nil))
                                 (list a (list b (list c d)))
16                               nil
                                 )))))
18    (defun get-subsumed-pr4 ()
      subsumed
20      )
    (defun get-subsuming-pr4  ()
22      subsuming
      )
24    (defun get-meronym-pr4  ()
      meronym
26      )
    (defun get-holonym-pr4  ()
28      holonym
      )
30    (defun proof-pr4 ()
      ;;;Call a generic function.
32      )
    (defun get-type-lst-pr4 ()
34      lst-types
      )
36    (defun  pr4 (smelly-square breeze-square gold-square)
      (funcall (eval constructor ) smelly-square breeze-square gold-square)))


1 ;;;; Created on 2008-10-24 16:30:42
  ;;;pr5.
3 (let  ( (name 'pr5)
          (lst-types '(next-position-is-a-valid-square smelly-square))
5         (subsumed '())
          (subsuming '())
7         (meronym '())
          (holonym '(next-position-is-a-valid-square smelly-square))
9         (constructor  '(lambda (x y)
                           (let*((a x)
11                               (b y)
                                 (c (funcall  (part-of) a b)))
13                           (if (not(equal c nil))
                                 (list a (list b c))
15                               nil
                                 )))))
17    (defun get-subsumed-pr5 ()
      subsumed
19      )
    (defun get-subsuming-pr5  ()
21      subsuming
      )
23    (defun get-meronym-pr5  ()
      meronym
25      )
    (defun get-holonym-pr5  ()
27      holonym
      )
29    (defun proof-pr5 ()
      ;;;Call a generic function.
31      )
    (defun get-type-lst-pr5 ()
33      lst-types
      )
35    (defun  pr5 (agent-direction-to-valid-square smelly-square)
      (funcall (eval constructor ) agent-direction-to-valid-square smelly-square)))


  ;;;; Created on 2008-10-18 15:30:36
2 ;;;pr1.
  (let ((name 'pr6)
```

```
        (lst-types '(smelly-square gold-square))
        (subsumed '())
        (subsuming '())
        (meronym '())
        (holonym nil)
        (constructor  '(lambda (x y)
                         (let*((a x)
                               (b y)
                               (c (funcall  (part-of) a b)))
                           (if (not(equal c nil))
                               (list a (list b c))
                               nil
                               )))))
  (defun get-subsumed-pr6 ()
    subsumed
    )
  (defun get-subsuming-pr6  ()
    subsuming
    )
  (defun get-meronym-pr6  ()
    meronym
    )
  (defun get-holonym-pr6  ()
    holonym
    )
  (defun proof-pr6 ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-pr6 ()
    lst-types
    )
  (defun  pr6 (smelly-square gold-square)
    (funcall (eval constructor ) smelly-square gold-square)))

;;;; Created on 2008-11-16 19:08:19
;;;; <>

(let ((name 'root)
        (lst-types '())
        (subsumed '())
        (subsuming '(agent-location))
        (meronym '())
        (holonym '())
        (current-value '())
        (level 2)
        (constructor  '()))

  (defun get-level-root()
    level
    )
  (defun initialise-root()
    (setf current-value '())
    )

  (defun get-subsumed-root ()
    subsumed
    )
  (defun get-subsuming-root  ()
    subsuming
    )
  (defun get-meronym-root  ()
    meronym
    )
  (defun get-holonym-root  ()
    holonym
    )
  (defun proof-root ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-root ()
    lst-types
    )
  (defun  root ()
    ()))

;;;; Created on 2008-10-25 11:10:35
;;;; Created on 2008-10-09 14:41:25
;;;; agent-has-gold

(let ((name 'agent-has-gold)
        (lst-types '(agent-location))
        (subsumed '(agent-location))
        (subsuming '())
        (meronym '())
        (holonym '())
        (current-value '())
        (level 2)
        (constructor  '(lambda (x)
                         (let*((a x)
                               (b (has-gold-p (pi-1 a) )))
```

```
16                          (if (and (not(equal b nil)))
                                (list a b)
18                              nil
                                )))))
20
  (defun get-level-agent-has-gold()
22   level
    )
24  (defun initialise-agent-has-gold()
    (setf current-value '())
26  )
  (defun get-subsumed-agent-has-gold ()
28   subsumed
    )
30  (defun get-subsuming-agent-has-gold  ()
    subsuming
32  )
  (defun get-meronym-agent-has-gold  ()
34   meronym
    )
36  (defun get-holonym-agent-has-gold  ()
    holonym
38  )
  (defun proof-agent-has-gold ()
40   ;;;Call a generic function.
    )
42  (defun get-type-lst-agent-has-gold ()
    lst-types
44  )
  (defun  agent-has-gold (agent-wump)
46   (funcall (eval constructor ) agent-wump)))

;;;; Created on 2008-10-09 14:39:41
2 ;;;agent-location (TDR).
(let ((name 'agent-location)
4         (lst-types '(agent-wumpus))
          (subsumed '())
6         (subsuming '(agent-has-gold ~agent-has-gold))
          (meronym '())
8         (holonym '())
          (current-value '())
10        (level 1)
          (constructor  '(lambda (x)
12                         (let*((a x)
                                 (b (agent-position a))
14                               (c (agent-direction a)))
                             (if (and (not(equal b nil))  (not(equal c nil)))
16                               (list a (list b c))
                                 nil
18                               )
                             ))))
20
  (defun get-level-agent-location()
22   level
    )
24  (defun initialise-agent-location()
    (setf current-value '())
26  )
  (defun execute-action-agent-location ()
28   (funcall action (pi-1 (pi-2 (pi-2 current-value ))))
    )
30  (defun get-subsumed-agent-location ()
    subsumed
32  )
  (defun get-subsuming-agent-location  ()
34   subsuming
    )
36  (defun get-meronym-agent-location  ()
    meronym
38  )
  (defun get-holonym-agent-location  ()
40   holonym
    )
42  (defun proof-agent-location ()
    ;;;Call a generic function.
44  )
  (defun get-type-lst-agent-location ()
46   lst-types
    )
48  (defun  agent-location (agent-wump)
    (funcall (eval constructor ) agent-wump)))
50
;;;(agent-location agent1)

1 ;;;; Created on 2008-10-09 15:15:04
  ;;;;  breeze-square
3 ;;;;(load "/home/patrick/Thesis/IntContextType/v02a/IntContextType002/IntContextType002/Ontologies/Contexts/DIR/tdr-breeze-

5 (let ((name 'breeze-square)
```

```
        (lst-types '(~agent-has-gold))
        (subsumed '(~agent-has-gold))
        (subsuming '())
        (meronym '(pr1 pr2 pr3 pr4))
        (holonym '())
        (current-value '())
        (level 3)
        (constructor  '(lambda (x)
                         (let*((a x)
                                (b (is-breezy-position-p (pi-1(pi-2(pi-1 a)))))
                               )
                           (if (and (not(equal b nil))
                                    )
                               (list a b)
                               nil
                               )))))
  (defun get-level-breeze-square()
    level
    )
  (defun initialise-breeze-square()
    (setf current-value '())
    )
  (defun get-subsumed-breeze-square ()
    subsumed
    )
  (defun get-subsuming-breeze-square  ()
    subsuming
    )
  (defun get-meronym-breeze-square ()
    meronym
    )
  (defun get-holonym-breeze-square  ()
    holonym
    )
  (defun proof-breeze-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-breeze-square ()
    lst-types
    )
  (defun  breeze-square (agent-location)
    (funcall (eval constructor ) agent-location)))


;;;; Created on 2008-10-09 15:22:53
;;;; gold-square
;;;;(load "/home/patrick/Thesis/IntContextType/v02a/IntContextType002/IntContextType002/Ontologies/Contexts/DIR/tdr-

(let ((name 'gold-square)
        (lst-types '(~agent-has-gold))
        (subsumed '(~agent-has-gold))
        (subsuming '())
        (meronym '(pr4))
        (holonym '())
        (current-value '())
        (level 3)
        (constructor  '(lambda (x)
                         (let*((a x)
                                (b (is-perceive-glitter-p (pi-1(pi-2(pi-1 a)))))
                               )
                           (if (and (not(equal b nil))
                                    )
                               (list a b)
                               nil
                               )))))
  (defun get-level-gold-square()
    level
    )
  (defun initialise-gold-square()
    (setf current-value '())
    )
  (defun get-subsumed-gold-square ()
    subsumed
    )
  (defun get-subsuming-gold-square  ()
    subsuming
    )
  (defun get-meronym-gold-square ()
    meronym
    )
  (defun get-holonym-gold-square  ()
    holonym
    )
  (defun proof-gold-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-gold-square ()
    lst-types
    )
  (defun  gold-square (agent-location)
```

```
    (funcall (eval constructor ) agent-location)))

;;;; Created on 2008-10-11 12:11:48
;;;; identify-dead-wumpus-square

(let ((name 'identify-dead-wumpus-square)
        (lst-types '(identify-wumpus-position wumpus))
        (subsumed '(identify-wumpus-position))
        (subsuming '())
        (meronym '())
        (holonym '())
        (current-value '())
        (level 5)
        (constructor  '(lambda (x y)
                         (let*((a x)
                                  (b y)
                                  (c (is-located-inp (pi-2(pi-1(pi-1 (a)))) b))
                                  (d (is-deadp b))
                               )
                           (if (not(equal c nil))
                               (list a(list b(list c d)))
                               nil
                               )))))
  (defun get-level-identify-dead-wumpus-square()
    level
    )
  (defun initialise-identify-dead-wumpus-square()
    (setf current-value '())
    )
  (defun get-subsumed-identify-dead-wumpus-square ()
    subsumed
    )
  (defun get-subsuming-identify-dead-wumpus-square  ()
    subsuming
    )
  (defun get-meronym-identify-dead-wumpus-square ()
    meronym
    )
  (defun get-holonym-identify-dead-wumpus-square  ()
    holonym
    )
  (defun proof-identify-dead-wumpus-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-identify-dead-wumpus-square ()
    lst-types
    )
  (defun  identify-dead-wumpus-square (identify-wumpus-position wumpus)
    (funcall (eval constructor ) identify-wumpus-position wumpus))
  )

;;;; Created on 2008-10-09 15:42:38
;;;; identify-wumpus-square

(let ((name 'identify-wumpus-square)
        (lst-types '(smelly-square position position smelly))
        (subsumed '(smelly-square))
        (subsuming '())
        (meronym '())
        (holonym '())
        (current-value '())
        (level 4)
        (constructor  '(lambda (v w x y)
                         (let*((a v)
                                  (b w)
                                  (c x)
                                  (d y)
                                  (e (is-adjacent-positionp (pi-1(pi-2(pi-1 a))) b))
                                  (f (next-position (pi-1(pi-2(a))) (pi-1(pi-2(pi-2(a)))) b))
                                  (g (is-adjacent-positionp b c))
                                  (h (is-adjacent-position b c))
                                  (i (is-smelly-positionp b d))
                                  (j (is-different-positionp (pi-1(pi-2(a))) c))
                               )
                           (if (and (not(equal e nil)) (not(equal f nil)) (not(equal g nil)) (not(equal h nil))
                                    (not(equal i nil)) (not(equal j nil))
                                    )
                               (list a(list b(list c(list d(list e(list f(list g(list h(list i j)))))))))
                               nil
                               )))))
  (defun get-level-identify-wumpus-square()
    level
    )
   (defun initialise-identify-wumpus-square ()
    (setf current-value '())
    )
  (defun get-subsumed-identify-wumpus-square ()
    subsumed
    )
  (defun get-subsuming-identify-wumpus-square  ()
```

```
    subsuming
    )
  (defun get-meronym-identify-wumpus-square ()
    meronym
    )
  (defun get-holonym-identify-wumpus-square  ()
    holonym
    )
  (defun proof-identify-wumpus-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-identify-wumpus-square ()
    lst-types
    )
  (defun  identify-wumpus-square (smelly-square position1 position2 smelly)
    (funcall (eval constructor ) smelly-square position1 position2 smelly)))
```

```
;;;; Created on 2008-10-25 11:41:31
;;;; next-position-is-a-valid-square

(let ((name 'next-position-is-a-valid-square)
      (lst-types '(~agent-has-gold))
      (subsumed '(~agent-has-gold))
      (subsuming '())
      (meronym '(pr1 pr5))
      (holonym '())
      (current-value '())
      (level 3)
      (constructor  '(lambda (x)
                       (let*((a x)
                             (b (is-possible-direction (pi-1(pi-2(pi-1 a)))  (pi-2(pi-2(pi-1 a))) ))
                         )
                         (if  (not(equal b nil))
                             (list a b)
                              nil
                              )))))
  (defun get-level-next-position-is-a-valid-square ()
    level
    )
  (defun initialise-next-position-is-a-valid-square()
    (setf current-value '())
    )
  (defun get-subsumed-next-position-is-a-valid-square ()
    subsumed
    )
  (defun get-subsuming-next-position-is-a-valid-square  ()
    subsuming
    )
  (defun get-meronym-next-position-is-a-valid-square ()
    meronym
    )
  (defun get-holonym-next-position-is-a-valid-square  ()
    holonym
    )
  (defun proof-next-position-is-a-valid-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-next-position-is-a-valid-square ()
    lst-types
    )
  (defun  next-position-is-a-valid-square (agent-location)
    (funcall (eval constructor ) agent-location)))
```

```
;;;; Created on 2008-10-09 14:41:25
;;;; ~agent-has-gold

(let ((name '~agent-has-gold)
      (lst-types '(agent-location))
      (subsumed '(agent-location))
      (subsuming '(next-position-is-a-valid-square breeze-square gold-square smelly-square ))
      (meronym '())
      (holonym '())
      (current-value '())
      (level 2)
      (constructor  '(lambda (x)
                       (let*((a x)
                             (b (~has-gold-p (pi-1 a))))
                         (progn
                         (if  (not(equal b nil))
                             (list a b)
                              nil
                              ))
                         )
                        )

                       ))
  (defun get-level-~agent-has-gold()
    level
    )
   (defun get-action-~agent-has-gold ()
```

```
    subsumed
29  )
  (defun initialise-~agent-has-gold()
31  (setf current-value '())
  )
33 (defun get-subsumed-~agent-has-gold ()
    subsumed
35  )
  (defun get-subsuming-~agent-has-gold  ()
37  subsuming
  )
39 (defun get-meronym-~agent-has-gold  ()
    meronym
41  )
  (defun get-holonym-~agent-has-gold  ()
43  holonym
  )
45 (defun proof-~agent-has-gold ()
    ;;;Call a generic function.
47  )
  (defun get-type-lst-~agent-has-gold ()
49  lst-types
  )
51 (defun  ~agent-has-gold (agent-wump)
    (funcall (eval constructor ) agent-wump)))

;;;; Created on 2008-10-25 11:34:58
2 ;;;; position-in-valid-square

4 (let ((name 'position-in-valid-square)
       (lst-types '(~agent-has-gold))
6      (subsumed '(~agent-has-gold))
       (subsuming '())
8      (meronym '(pr1))
       (holonym '())
10     (current-value '())
       (constructor  '(lambda (x)
12                      (let*((a x)
                              (b (s-valid-position-p (pi-1(pi-2(pi-1 a)))) (pi-2(pi-2(pi-1 a))))
14                      )
                        (if  (not(equal b nil))
16                           )
                          (list a b)
18                        nil
                          ))))
20 (defun get-level-position-in-valid-square()
    level
22  )
  (defun initialise-position-in-valid-square()
24  (setf current-value '())
  )
26 (defun get-subsumed-position-in-valid-square ()
    subsumed
28  )
  (defun get-subsuming-position-in-valid-square  ()
30  subsuming
  )
32 (defun get-meronym-position-in-valid-square ()
    meronym
34  )
  (defun get-holonym-position-in-valid-square  ()
36  holonym
  )
38 (defun proof-position-in-valid-square ()
    ;;;Call a generic function.
40  )
  (defun get-type-lst-position-in-valid-square ()
42  lst-types
  )
44 (defun  position-in-valid-square (~agent-has-gold)
    (funcall (eval constructor ) ~agent-has-gold)))

1 ;;;; Created on 2008-10-09 15:34:48
  ;;;; smelly-square
3
  (let ((name 'smelly-square)
5       (lst-types '(~agent-has-gold))
       (subsumed '(~agent-has-gold))
7      (subsuming '(identify-wumpus-square))
       (meronym '(pr3 pr4 pr5 pr6))
9      (holonym '())
       (current-value '())
11     (level 3)
       (constructor  '(lambda (x)
13                      (let*((a x)
                              (b (is-smelly-position-p (pi-1(pi-2(pi-1 a)))))
15                      )
                        (if  (not(equal b nil))
17                           (list a b)
                             nil
```

```
                                )
                            ))))
  (defun get-level-smelly-square()
    level
    )
(defun initialise-smelly-square()
    (setf current-value '())
    )
  (defun get-subsumed-smelly-square ()
    subsumed
    )
  (defun get-subsuming-smelly-square  ()
    subsuming
    )
  (defun get-meronym-smelly-square ()
    meronym
    )
  (defun get-holonym-smelly-square  ()
    holonym
    )
  (defun proof-smelly-square ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-smelly-square ()
    lst-types
    )
  (defun  smelly-square (~agent-has-gold)
    (funcall (eval constructor ) ~agent-has-gold)))

;;;; Created on 2008-10-25 11:57:36
;;;ded-01.
;;;Permet de déterminer que la case suivante est une case valide.
(let ((lst-types '(position position position))
        (subsumed '())
        (subsuming '())
        (meronym '())
        (holonym '(agent-direction-to-valid-square breeze-square))
        (action '())
        (current-value '())
        (constructor  '(lambda (x y z)
                         (let*((a x)
                               (b y)
                               (c z)
                               (d (resolution a b c))
                           )
                           (if (not (equal d nil))
                               (list a (list b c))
                               nil))))
      )
  (defun initialise-ded-01()
    (setf current-value '())
    )
  (defun get-current-value-ded-01()
    current-value
    )
  (defun get-action-ded-01 ()
    action
    )
  (defun execute-action-ded-01 ()
     nil
    )
  (defun get-subsumed-ded-01 ()
    subsumed
    )
  (defun get-subsuming-ded-01  ()
    subsuming
    )
  (defun get-meronym-ded-01  ()
    meronym
    )
  (defun get-holonym-ded-01  ()
    holonym
    )
  (defun proof-ded-01 ()
    ;;;Call a generic function.
    )
  (defun get-type-lst-ded-01 ()
    lst-types
    )
  (defun  ded-01 (position1 position2 position3)
   (funcall (eval constructor ) position1 position2 position3)
  )
  )
;;;Example
;;(get-lst-types-of-sigma-type 'ded-01)
;;(setf  (gethash 'position agent-kb) (list position11 position12 position22))
;;(setf (gethash 'adjacent-east agent-kb) (list (list position12 position22)))
;;(setf (gethash 'adjacent-north agent-kb) (list (list position11 position12)))
;;(setf (gethash 'adjacent-south agent-kb) (list (list position12 position11)))
```

```
;;(setf (gethash 'adjacent-west agent-kb) (list (list position22 position12)))
;;(setf lst-of-extension  (get-extension-of-types-lst (get-lst-types-of-sigma-type 'ded-01)))
;;(setf lst-of-cartesion-product-of-extension (cartesian-product lst-of-extension))
;;(get-proof-of-tdr 'ded-01 lst-of-cartesion-product-of-extension)
;;(ded-01 te1 position22)

;;(setf te1 (first (first lst-of-cartesion-product-of-extension)))
;;(setf b1 (next-position-kb (first (second te1)) (second (second te1))))

;;(setf te2 (first (get-current-value-ded-01)))
;;(setf te3 '(set-position-valid))
;;(funcall (first te3) (first(second(second te2))))

;;(agent-direction (first (first lst-of-cartesion-product-of-extension) ))
;;;;;;(get-lst-proof-of-tdr 'agent-location);;Ok
;;(get-lst-proof-of-tdr 'ded-01)
```

```
;;;; Created on 2008-09-28 12:17:19
;;;;(load "/home/patrick/Thesis/IntContextType/v02a/IntContextType002/IntContextType002/pi-wumpus-01.lisp")
(let ((pi-types '()))
  (defun get-pi-ontologie-lst ()
    (second pi-types))
  (defun add-pi-type (pi-value)
    (push pi-value pi-types)
    )
  (defun pi-types-p (value)
    (member value pi-types)
    )
  )

(deftype pi-types ()  '(satisfies pi-types-p))

;;;agent-position (sigma).
(add-pi-type 'agent-position)

(let ((lst-types '(agent-wumpus))
      (type0 pi-type)
       (constructor '(lambda (x) (positionG x))))
  (defun proof-agent-location ()
    ;;;Call a generic function.
    )
  (defun get-agent-position-lst ()
    lst-types
    )
  (defun  agent-position1 ()
    (eval constructor)))

;;;agent-direction (sigma).
(add-pi-type 'agent-direction)

(let ((lst-types '(agent-wumpus))
       (constructor '(lambda (x) (direction x))))
  (defun proof-agent-direction ()
    ;;;Call a generic function.
    )
  (defun get-agent-direction-lst ()
    lst-types
    )
  (defun  agent-direction1 ()
    (eval constructor)))
```

```
;;;; Created on 2008-09-28 11:33:39
;;;;(load "/home/patrick/Thesis/IntContextType/v02a/IntContextType002/sigma-wumpus-01.lisp")

(deftype sigma-types ()  '(satisfies sigma-types-p))

;;;Example :
;;(add-sigma-types-extension 'agent-location (get-lst-proof-of-tdr 'agent-location))
;;(typep 'agent-location 'sigma-types)
```

Index

Index

Bibliographie

[1] V. Akman and M. Surav. The use of situation theory in context modeling. *Computational Intelligence*, 12(4) :1–13, 1996.

[2] Anonymous. The Common Lisp Object Standard (CLOS), October 1987. 1 videocassette (VHS) (53 min.).

[3] R. Artale, E. Franconi, and N. Guarino. Open problems with part-whole relations. In *In Proc. of the 1996 Description Logic Workshop (DL-96), number WS-96-05*, pages 70–73. AAAI Press, 1996.

[4] Nicholas Asher and Alex Lascarides. *Logics of Conversation (Studies in Natural Language Processing)*. Cambridge University Press, June 2005.

[5] R. Baeza-Yates. Average running time of the boyer-moore-horspool algorithm. *Theoretical Computer Science*, 92(1) :19–31, January 1992.

[6] H. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, 1992.

[7] H. Barendregt and H. Geuvers. *Handbook of Automated Reasoning*, chapter Proof-Assistants Using Dependent Type Systems, pages 1149–1238. Elsevier and MIT Press, 2001.

[8] H.P Barendregt. Introduction to generalized type systems. *Journal of functional programming*, vol. 1 :124 – 154, 1991.

[9] P. Barlatier and R. Dapoigny. Using contexts to prove and share situations. In *FLAIRS Conference*, pages 448–453, 2007.

[10] P. Barlatier and R. Dapoigny. A theorem prover with dependent types for reasoning about actions. In *STAIRS'08 Procs.*, pages 12–23, 2008.

[11] B Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *FoSSaCS*, pages 365–379, 2008.

[12] J. Barwise. *On Conditionals*, chapter Conditionals and conditional information, pages 21–54. Cambridge University Press, 1986.

[13] E. Benoit. *Capteurs symboliques et capteurs flous : un nouveau pas vers l'intelligence*. PhD thesis, Université Joseph Fourier, 1993.

[14] G. Betarte. Type checking dependent (record) types and subtyping. *Journal of Functional and Logic Programming*, 10(2) :137–166, 2000.

[15] T. Bittner and M. Donnelly. Computational ontologies of parthood, componenthood, and containment. In *IJCAI*, pages 382–387, 2005.

[16] Kamareddine F. Borghuis, T. and R. Nederpelt. Formalizing belief revision in type theory. *Journal of the IGPL*, 10(5) :461–500, 2002.

[17] Alexander Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1–2) :353–367, 1996.

[18] P. Bouquet, L. Serafini, P. Brézillon, M. Benerecetti, and F. Castellani, editors. *Modelling and Using Context*, number 1688 in LNCS. Springer Verlag, 1999.

[19] R. J. Brachman. On the epistemological status of semantic networks. In N. V. Findler, editor, *Associative Networks : Representation and Use of Knowledge by Computers*, pages 3–50. Academic Press, Orlando, 1979.

[20] P. Brézillon. Hors du contexte and point de salut. Séminaire 'Objets Communicants', Autrans, France, 2002.

[21] P Brézillon. Context dynamic and explanation in contextual graphs. In *Modeling and Using Context (CONTEXT-03*, pages 94–106. Springer Verlag, 2003.

[22] P. Brézillon, L. Pasquier, and J.C. Pomerol. Reasoning with contextual graphs. *European Journal of Operational Research*, 136(2) :290–298, 2001.

[23] Patrick Brézillon and Marcos Cavalcanti. International and interdisciplinary conference context-97. 2007.

[24] S. Buvac. Quantificational logic of context. In *Procs. of the 13th National Conference on Artificial Intelligence*, volume 1, pages 600–606, 1996.

[25] S. Buvac, V. Buvac, and I. A. Mason. Metamathematics of contexts. *Fundamentae Informaticae*, 23(3) :412–419, 1995.

[26] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, R. Rosati, D. Calvanese, M. Lenzerini, D. Nardi, and R. Rosati. Description logics for information integration. In *In Computational Logic : From Logic Programming into the Future (In honour of Bob Kowalski), Lecture Notes in Computer Science*, pages 41–60. Springer, 2001.

[27] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Description logics for conceptual data modeling. *Logics for databases and information systems*, pages 229–263, 1998. Kluwer Academic Publishers, Norwell, MA, USA.

[28] P. Cartier. *Logique, catégories et faisceaux.* Séminaire Bourbaki, 20 (1977-1978), Exposé No. 513, 24 p., 1978.

[29] M. Chalmers. A historical view of context. *Computer supported cooperative work*, 13(3) :223–247, 2004.

[30] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Department of Computer Science, Dartmouth College, 2000.

[31] H. Chen, T. Finin, and A. Joshi. Using owl in a pervasive computing broker. In *Procs. of Workshop on Ontologies in Open Agent Systems (AAMAS'03)*, 2003.

[32] K. Cheverst, K. Mitchell, and N. Davies. 1 design of an object model for a context sensitive tourist guide.

[33] N.B. Cochiarella. Logic and ontology. *Axiomathes*, 12 :117–150, 2001.

[34] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[35] R. Cooper. Records and record types in semantic theory. *J. Log. Comput.*, 15(2) :99–112, 2005.

[36] T. Coquand and G. Huet. Constructions : A higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra (1)*, pages 151–184, 1985.

[37] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation 76 (88), no. 2-3*, pages 95–120, 1988.

[38] J. Coutaz, J. Crowley, S. Dobson, and D. Garlan. Context is key. *Communications of the ACM*, 48(3) :49–53, 2005.

[39] J.L. Crowley, J. Coutaz, G. Rey, and P. Reignier. Perceptual components for context aware computing. In *UbiComp 2002*.

[40] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[41] R. Dapoigny and P. Barlatier. Deriving behavior from goal structure for the intelligent control of physical systems. In J.A. Cetto J.Filipe, J.L. Ferrier and M. Carvalho, editors, *Informatics in Control, Automation and Robotics II*, volume 15, pages 51–58. Springer, 2007.

[42] R. Dapoigny and P. Barlatier. Goal reasoning with context record types. In *CONTEXT*, pages 164–177. Springer Berlin / Heidelberg, 2007.

[43] R. Dapoigny and P. Barlatier. Causal reasoning with contexts using dependent types. In *FLAIRS Conference*, pages 107–108, 2008.

[44] R. Dapoigny and P. Barlatier. Towards a conceptual structure based on type theory. In *ICCS Supplement*, pages 107–114, 2008.

[45] R. Dapoigny and P. Barlatier. Vers un modèle formel pour le raisonnement à partir des contextes. *Revue d'Intelligence Artificielle*, 22(6) :725–755, 2008.

[46] R. Dapoigny and P. Barlatier. Reasoning about relations with dependent types : Application to context-aware applications. In *ISMIS'09 Procs*. Springer, 2009.

[47] R. Dapoigny and P. Barlatier. Towards an ontological modeling with dependent types : Application to part-whole relations. In *ER'09 Procs*. Springer, 2009.

[48] G. De Lavalette and R. Renardel. Strictness analysis via abstract interpretation for recursively defined types. *Inf. Comput.*, 99(2) :154–177, 1992.

[49] Maria Stela V. de Paiva. Constructive description logics : what, why and how. Workshop on Context Representation and Reasoning (CRR 2006).

[50] M. Dean and Guus Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004.

[51] N.G. deBruijn. *The Mathematical Language AUTOMATH, Its Usage and Some of its Extensions*. Springer, 1970.

[52] K. Devlin. *Logic and Information Theory*. Cambridge University Press, 1991.

[53] Dey and K. Anind. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1) :4–7, 2001.

[54] AnindK̃. Dey and Abowd Gregory D. Towards a better understanding of context and context-awareness. In *Procs. of the CHI 2000 Workshop on The What, Who, Where, When, and How of Context-Awareness*, 2000.

[55] P. Dockhorn Costa, J. P. A. Almeida, L. F. Pires, G. Guizzardi, and M. van Sinderen. Towards conceptual foundations for context-aware applications. In *Procs. of the AAAI'06 Workshop on Modeling and Retrieval of Context*, pages 54–58. AAAI Press, 2006.

[56] P. Dourish. Seeking a foundation for context-aware computing. *Human-Computer Interaction*, 16(2-3), 2001.

[57] P. Dourish. What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 8(1) :19–30, 2004.

[58] J. Euzenat, J Pierson, and F. Ramparany. Dynamic context management for pervasive applications. *Knowl. Eng. Rev.*, 23(1) :21–49, 2008.

[59] L. Ferreira Pires, M. van Sinderen, E. Munthe-Kaas, S. Prokaev, Hutschemaekers M., and D.-J. Plas. Techniques for describing and manipulating context information. Freeband/A_MUSE D3.5v2.0, Lucent Technologies, 2005.

[60] A. Gangemi and P. Mika. Understanding the semantic web through descriptions and situations. In *International Conference on Ontologies, Databases and Applications of SEmantics (ODBASE 2003)*, Catania, (Italy), November 2003.

[61] C. Ghidini and F. Giunchiglia. Local models semantics, or contextual reasoning = locality + compatibility. *Artificial Intelligence*, 127(2) :221–259, 2001.

[62] J. Girard. *Le lambda-calcul du second ordre.* Séminaire Bourbaki, 29 (1986-1987), Exposé No. 678, 13 p., 1987.

[63] J.Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de L'arithmétique D'ordre Supérieur.* Thèse de Doctorat, Université Paris VII, 1972.

[64] F. Giunchiglia. Towards a logical treatment of qualitative reasoning. In *Procs. of the 1rst workshop on Qualitative Reasoning about Physical Systems*, 1991.

[65] F. Giunchiglia. Contextual reasoning. Technical Report 9211-20, Istituto per la Ricerca Scientifica e Technologica, 1992.

[66] A. Göker. Capturing information need by learning user context. In *IJCAI Workshop on Learning about users*, 1999.

[67] C. Golbreich, O. Bierlaire, O. Dameron, and B. Gibaud. Use case : Ontology with rules for identifying brain anatomical structures. In *W3C Workshop on Rule Languages for Interoperability*, 2005.

[68] Paul Graham. *On Lisp : Advanced Techniques for Common Lisp.* Prentice Hall PTR, 1993. http ://www.paulgraham.com/onlisp.html.

[69] Thomas R. Gruber. Ontolingua : A mechanism to support portable ontologies. Technical report, 1992.

[70] N. Guarino. Formal ontology and informations systems. In *Procs. of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98)*, pages 3–15, 1998.

[71] N. Guarino, M. Carrara, and P. Giaretta. Formalizing ontological commitments. In *AAAI '94 : Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 560–567, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[72] K. Hanna, N. Daeche, and G. Howells. Implementation of the veritas design logic. pages 77–94, 1990.

[73] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Procs. of the 21st Symposium on Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[74] Jerry R. Hobbs. Ontological promiscuity. In *Proceedings, 23rd Annual Meeting of the Association for Computational Linguistics*, pages 61–69, 1985.

[75] I. Horrocks and F. Patel-Schneider. A proposal for an owl rules language. In *Procs. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731, 2004.

[76] W. A. Howard. *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, chapter The formulae-as-types notion of construction, pages 479–490. Academic Press, 1980.

[77] G. Huet and T. Coquand. *Coquand Huet : Constructions : a higher order proofs system for mechanizing mathematics*. Technical report, INRIA, 401 edition, May 1985.

[78] G. Huet and C. Paulin-Mohring. *Preuves et constructions de programmes = Proof and program construction*. CNRS, Paris, FRANCE (Revue), 1993.

[79] J. Indulska, R. Robinson, A. Rakotonirainy, and K. Henricksen. Experiences in using cc/pp in context-aware systems. In *In Proc. of the Intl. Conf. on Mobile Data Management (MDM*, pages 247–261. Springer, 2003.

[80] B. Jacobs and T.F. Melham. Translating dependent type theory into higher order logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA'93)*, pages 209–229, 1993.

[81] M. Kaenampornpan and E. O'Neill. Modelling context : An activity theory approach. In *in Ambient Intelligence : Second European Symposium, EUSAI 2004*, pages 367–374. Springer, 2004.

[82] H. Kamp and U. Reyle. A calculus for first order discourse representation structures. *Logic, Language and Information*, 1996.

[83] A. Kofod-Petersen and J. Cassens. Using activity theory to model context awareness. In *Procs. of Modeling and Retrieval of Context (MRC2005)*, pages 1–17, 2005.

[84] A. Kopylov. Dependent intersection : A new way of defining records in type theory. In *Procs. of the 18th An. IEEE Symposium on Logic in Computer Science*, pages 86–95, 2003.

[85] G. Kreisel. *La prédicativité*. Bulletin de la S.M.F., tome 88, 1960.

[86] Douglas B. Lenat, C. Sierra, R.V. Guha, K. Pittman, D. Pratt, and M. Shepherd. Cyc : toward programs with common sense. *Communications of the ACM*, 33(8) :30–49, 1990.

[87] G. Löhrer. On ends and means. constructive type theory as a guide for modeling in theory of mind and action. *Mathematics and Social Sciences*, 171(3) :5–23, 2005.

[88] S. W. Loke. Representing and reasoning with situations for context-aware pervasive computing : a logic programming perspective. *The Knowledge Engineering Review*, 19 :213–233, 2004.

[89] J. R. Longley and A. K. Simpson. A uniform approach to domain theory in realizability models. *Mathematical. Structures in Comp. Sci.*, 7(5) :469–505, 1997.

[90] Z. Luo. A unifying theory of dependent types i. LFCS Report Series ECS-LFCS-91-154, University of Edinburgh, 1991.

[91] Z. Luo. A unifying theory of dependent types : The schematic approach. In *Procs. of Logical Foundations of Computer Science (LFCS'92)*, pages 293–304, 1992.

[92] Z. Luo. *Computation and Reasoning : A Type Theory for Computer Science*. Oxford University Press, 1994.

[93] Z. Luo. Manifest fields and module mechanisms in intensional type theory. In *Procs. of TYPES'08*, 2008.

[94] Z. Luo and R. Pollack. *LEGO proof development system : User's manual.* Computer Science Dept., University of Edinburgh, 1992.

[95] M. Luther, Y. Fukazawa, M. Wagner, and S. Kurakake. Situational reasoning for task-oriented mobile service recommendation. *The Engineering Review*, 23(1) :7–19, 2008.

[96] L. Magnusson and B. Nordstrom. The alf proof editor and its proof engine. 1994.

[97] C. Maria Keet. Part-whole relations in object-role models. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (2)*, volume 4278 of *Lecture Notes in Computer Science*, pages 1118–1127. Springer, 2006.

[98] P. Martin-Löf. Constructive mathematics and computer programming. *Logic, Methodology and Philosophy of Sciences*, 6 :153–175, 1982.

[99] S. Matwin and M. Kubat. The role of context in concept learning. In *ICML Workshop on Learning in Context-sensitive Domains*, pages 1–5, 1996.

[100] J. McCarthy. Notes on formalizing context. In *Procs. of the 13th Int. Joint Conf. on Artificial Intelligence*, pages 555–560, 1993.

[101] M. Minsky. *A Framework for Representing Knowledge.* The Psychology of Computer Vision, P. Winston (Ed.), McGraw-Hill, 1975, 1974.

[102] D. Miéville. *Un développement des systèmes logiques de Stanislaw Lesniewski.* Peter Lang, 1984.

[103] R. Montague. Pragmatics and intensional logic. *Synthèse*, 22 :68–94, 1970.

[104] Amedeo Napoli. Subsumption and classification-based reasoning in object-based representations. In *ECAI*, pages 425–429, 1992.

[105] F. Nef. *Recherches sur l'ontologie de l'objet*, volume 44 of *Problèmes & Controverses.* Vrin, 1998.

[106] D. Nicklas, M. Grossmann, J. Minguez, and M. Wieland. Adding high-level reasoning to efficient low-level context management : A hybrid approach. In *Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 447–452. IEEE Computer Society, 2008.

[107] Lassila Ora and Swick Ralph R. Resource description framework (rdf) model and syntax specification. World Wide Web Consortium, Recommendation REC-rdf-syntax-19990222, February 1999.

[108] N. Oury and W. Swierstra. The power of pi. *SIGPLAN Notices*, 43(9) :39–50, 2008.

[109] S. Owre, J.M. Rushby, N. Shankar, and F. Von Henke. Formal verification for fault-tolerant architectures : Prolegomena to the design of pvs. pages 107–125, 1995.

[110] J. Pascoe. Adding generic contextual capabilities to wearable computers. 1998.

[111] Anca PASCU. *Modèles logico-mathématiques en linguistique.* Thèse de Doctorat, Université Paris IV, 2001.

[112] R. Pollack. Introduction dependently typed records for representing mathematical structure, 2000.

[113] R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13(3-5) :386–402, 2002.

[114] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International Semantic Web Conference*, pages 30–43, 2006.

[115] M. Rehak, M. Gregor, M. Pechoucek, and J.M. Bradshaw. Representing context for multiagent trust modeling. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 737–746, 2006.

[116] H. Reichgelt, P. Jackson, and F. V. Harmelen. *Logic-based knowledge representation.* MIT Press, 1989.

[117] J.C. Reynolds. *Towards a Theory of Type Structure.* Springer-Verlag, 1974.

[118] C. Roche. *Terminologie et ontologie.* Larousse, 2005.

[119] A. N. Russell, B. et Whitehead. *Principia Mathematica.* 1910.

[120] B. Russell. *Mathematical logic as based on the theory of types.* 1908.

[121] S. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[122] W. K. Chan Samuel and F James. Dynamic context generation for natural language understanding : A multifaceted knowledge approach. *IEEE Transactions On Systems, Man, And Cybernetics (Part A)*, 33(1) :23–41, 2003.

[123] U. Sattler. A concept language for an engineering application with part-whole relations. In *Procs. of the international workshop on description logics*, pages 119–123, 1995.

[124] B. N. Schilit, N. L. Adams, and R. Want. Context-aware computing applications. 1994.

[125] A. Schmidt, M. Beigl, and H. Gellersen. There is more to context than location. *Computers and Graphics*, 23 :893–901, 1999.

[126] P. Simons. *Parts : A Study in Ontology.* Clarendon Press, 2000.

[127] B. Smith. *The Basic Tools of Formal Ontology.* Formal Ontology in Information Systems Amsterdam, Oxford, Tokyo, Washington, DC : IOS Press, 1998.

[128] B. Smith and D. Mark. Geographic categories : An ontological investigation. *International Journal of Geographical Information Science*, 15 :591–612, 2001.

[129] J. Sowa. *Knowledge Representation : Logical, Philosophical, and Computational Foundations.* China Machine Press, 2000.

[130] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *Sixth International Conference on Ubiquitous Computing (UbiComp2004)*, pages 34–41, 2004.

[131] M. Surav and V. Akman. Modeling context with situations. Technical report, 1995.

[132] M. Surav and V. Akman. Modelling context with situation. Technical Report BU-CEIS-95-07, Department of Computer Engineering and Information Science, Bilkent University, 1995.

[133] Sakari Tamminen, Antti Oulasvirta, Kalle Toiskallio, and Anu Kankainen. Understanding mobile contexts. *Personal Ubiquitous Comput.*, 8(2) :135–143, 2004.

[134] M. Thielscher. A flux agent for the wumpus world. In Proceedings of the Workshop on Nonmonotonic Reasoning, Action and Change (NRAC2005), Edinburgh, UK, 2005.

[135] M. Thielscher. Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.*, 2 :179–192, 1998.

[136] M. Thielscher. Flux : A logic programming method for reasoning agents. *CoRR*, cs.AI/0408044, 2004.

[137] R. Thomason. Representing and reasoning with context. In *Procs. of the International Conference on Artificial Intelligence and Symbolic Computation*, number 1476 in LNCS, pages 29–41, 1998.

[138] R. Thomason. Type theoretic foundations for context, part 1 : Contexts as complex type-theoretic objects. In *CONTEXT*, number 1688 in LNCS, pages 351–360. Springer, 1999.

[139] M. Uschold and M. Grüninger. Ontologies : Principles, methods and applications. *The Knowledge Engineering Review*, 11(2) :93–155, 1996.

[140] Saba Walid. Language, logic and ontology : uncovering the structure of commonsense knowledge, 2007. In Press, International Journal of Human-Computer Studies.

[141] M. E. Winston, R. Chaffin, and D. Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 1(1) :417–444, 1987.

[142] Wang Xiao H., Gu Tao, Zhang Da Q., and Pung Hung K. Ontology based context modeling and reasoning using owl. In *Procs. of the 2nd IEEE Conference on Pervasive Computing and Communications (PerCom2004)*, pages 18–22, 2004.

[143] K. Yang and A. Galis. Policy-driven mobile agents for context-aware service in next generation networks. In *MATA*, pages 111–120, 2003.

Résumé

Le résumé.

Mots-clés: Contextes, Raisonnement, Théorie Constructive des Types, Types Dépendants, Sous-typage, Vérification de Type, Ontologies.

Dans ce mémoire, nous proposons une solution possible à la question suivante : comment formaliser des environnements associés à un processus (quelconque) et comment utiliser les informations qu'ils contiennent pour produire des actions pertinentes ? Cette question nous a amené à introduire la notion de contexte ainsi qu'une représentation réutilisable des connaissances pour le formaliser. Nous nous sommes donc intéressés aux notions d'ontologies, de contextes et d'actions. Pour la représentation et le raisonnement sur les contextes et les actions, nous proposons une solution appelée DTF. Celle-ci étend une théorie constructive des types existante permettant ainsi de disposer d'une grande expressivité, de la décidabilité de la vérification de types et d'un mécanisme de sous-typage efficace. Nous montrons comment modéliser les contextes et les actions sous la forme de types dépendants à partir des données fournies sur un problème et des actions à entreprendre pour le résoudre. Enfin, pour tester la faisabilité et pouvoir juger de la complexité d'une telle solution, un "démonstrateur de contexte " est réalisé avec un langage fonctionnel. Puis, une application test appelée " le monde du Wumpus " où un agent logiciel se déplace dans un environnement inconnu, est alors implantée en LISP.

Abstract

Keywords: Contexts, Actions, Reasoning, Constructive Type Theory, Dependent Types, Subtyping, Type checking, Ontologies.

This approach suggests a possible solution for the following issue : how to formalize environments related to a given process and how to exploit the information they provide in order to start appropriate actions ? For that purpose, we took an interest in the concepts of ontologies, contexts and actions. We have investigated the constructive type theory and extended it with extensional sub-typing (allowing for type hierarchies) and constants to result in what is called the Dependent Type Framework (DTF). DTF tries to combine a constructive logic with a functional programming language for the representation and reasoning about contexts and actions. It provides a high expressiveness, decidability of type checking and a powerful sub-typing mechanism. We show how to model both contexts with types in DTF from the given information about a problem and actions to start to solve it. Then, as a test of feasibility with the purpose of sensing the complexity of such a solution, a context prover has been built with a functional language. Finally, a test application called the wumpus world in which a software agent moves across a grid in an unknown environment is implemented.