MASTER EN SCIENCES INFORMATIQUES

L'héritage dans les langages de programmation selon une perspective ontologique

VEULLIET, Sylvain

Award date:
2023

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022-2023

L'héritage dans les langages de programmation selon une perspective ontologique

VEULLIET Sylvain

..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : ENGLEBERT Vincent

Co-promoteur : VANHOOF Wim

Mémoire présenté en vue de l'obtention du grade de Master 60 en Sciences Informatiques

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 108 B-5000 NAMUR(BELGIUM)

Remerciements

Je tiens tout d'abord à remercier ma famille qui m'a soutenu durant la réalisation de ce travail et de mes études en général.

Je souhaite également remercier mon promoteur V. Englebert et mon co-promoteur W. Vanhoof pour les conseils qu'ils m'ont prodigués ainsi que pour le suivi continu de ce mémoire.

Résumé

Les langages de programmation supportant le paradigme orienté objet possèdent de nombreuses similarités dont l'utilisation de l'héritage. La manière dont l'héritage est implémenté dans chaque langage peut varier. Il existe divers moyens de pouvoir comparer ces implémentations, l'utilisation d'ontologies est un de ces moyens. Les ontologies permettent de représenter la connaissance sur un domaine et de pouvoir questionner des raisonneurs, notamment lorsqu'elles reposent sur les logiques de description.

Dans le cadre de ce mémoire, l'héritage dans plusieurs langages est étudié et représenté en utilisant des ontologies. Pour arriver à ce résultat, chaque langage fut analysé et un méta-modèle reprenant les divers concepts tirés de cette analyse a été réalisé. Les ontologies produites permettent de mettre en avant les ressemblances et les différences entre les langages étudiés, parfois même entre deux versions d'un même langage. Chaque langage possède sa propre ontologie, qui se base sur une ontologie commune définissant les divers concepts rencontrés lors de l'analyse.

Mots-clés *: héritage, ontologie, langage de programmation, orienté objet*

Abstract

Programming languages supporting the object-oriented paradigm have multiple similarities. Amongst those similarities, there's the support of inheritance. The way inheritance is implemented in each language may vary. There are various ways of comparing those implementations, using ontologies is one of them. Ontologies are extremely useful when it comes to representing knowledge about a domain and to be able to use reasoners, especially when they rely on description logics.

In the context of this memoir, inheritance in multiple languages has been studied and represented using ontologies. To achieve this result, each language has been analyzed and a meta-model incorporating the various concepts drawn from this analysis was carried out. The ontologies produced allow to highlight the differences and the similarities between the studied languages, sometimes even between two versions of the same language. Each language owns its own ontology which is based on a common shared ontology describing the concepts found during the analysis.

Keywords *: inheritance, ontology , programming language, object-oriented*

Table des matières

1 Introduction

1.1 Contexte

On dit souvent qu'il existe autant de langages de programmation que de langages parlés. Cet adage, bien qu'extrêmement imprécis, met en avant le nombre important de différents langages de programmation. Ces langages peuvent être regroupés de diverses manières, que ce soit par les objectifs et les buts pour lesquels ils ont été développés, par les différents paradigmes qu'ils supportent, les grands concepts qu'ils utilisent, etc. Parmi ces concepts se trouve la notion d'héritage. L'héritage est fortement lié au paradigme orienté objet et bien que ce concept date de plusieurs décennies, il n'est pas aisé de trouver une définition qui mette tout le monde d'accord. En effet, comme présenté par [56], il existe plusieurs types d'héritage, servant des buts différents. On peut citer les trois types principaux qui en ressortent : la spécialisation, le sous-classement et le sous-typage. Ces notions seront développées plus en avant dans ce travail.

Chaque langage de programmation peut implémenter l'héritage de manière arbitraire, avec plus ou moins de flexibilité. Il serait intéressant de pouvoir comparer les sémantiques de ces langages, de mettre en avant les ressemblances et les différences. Il existe plusieurs moyens de comparer des éléments entre eux, l'un de ces moyens est d'utiliser une base de connaissance qui contiendra les informations propres à chaque langage. Une telle base de connaissance peut par exemple être exprimée via une ontologie. Les ontologies permettent de représenter formellement un domaine (ou une partie délimitée de ce dernier) spécifique, les différents concepts qui le composent ainsi que les relations entre ces concepts. De plus, avec l'avènement du web sémantique, l'utilisation d'ontologies devient encore plus intéressant puisqu'elles pourront être échangées, exploiter les ressources disponibles sur le net de manière automatisée mais également organiser l'information de manière formelle.

1.2 Question de recherche

L'objectif de ce mémoire est donc de se servir des ontologies pour représenter comment l'héritage fonctionne dans différents langages de programmation. Ceci nous amène alors à la question de recherche suivante : "Comment représenter le concept de l'héritage dans les langages de programmation selon une perspective ontologique ?".

1.3 Méthodologie

Comme expliqué précédemment, vu le nombre important de langages de programmation, il a fallu sélectionner certains langages qui seront les sujets d'étude pour ce mémoire. Les langages choisis l'ont été selon plusieurs critères : la pertinence, la popularité et les possibles particularités qu'ils possèdent. Le choix de se focaliser sur l'héritage a été fait car c'est une notion importante dans les langages usant du paradigme orienté objet et ce dernier fait partie des paradigmes les plus utilisés dans le monde. Sur base de ces critères, et en accord avec mes promoteurs, les langages suivants ont été retenus :

1. Simula : Il est le premier langage orienté objet et apporte plusieurs notions dont celle de l'héritage. Il est intéressant de savoir à quoi ressemblait l'héritage à ce moment-là.
2. Java : Java fait partie des langages les plus utilisés au monde et est orienté objet.

3. C# : Un autre langage populaire également orienté objet.
4. F# : Langage appartenant à la même famille que C# avec ses particularités propres comme le fait de supporter plusieurs paradigmes de programmation.
5. Scala : Langage proche de Java avec quelques particularités, dont le support du paradigme fonctionnel.
6. C++ : Langage populaire permettant d'utiliser l'héritage multiple.
7. OCaml : Langage atypique, par rapport aux autres, usant de plusieurs paradigmes et d'héritage multiple.
8. Haskell : Langage non orienté objet et purement fonctionnel, il est intéressant de voir si l'héritage peut être simulé d'une quelconque manière.

Pour juger de la popularité d'un langage, il existe différents moyens. Certains sites internet proposent des classements réalisés sur base de critères divers et variés. C'est par exemple le cas du site "PYPL PopularitY of Programming Language" [1] qui classe les langages les plus populaires en fonction de la fréquence de recherche de tutoriels sur Google.

Avoir sélectionné ces 8 langages permet d'avoir un échantillon suffisamment large pour réaliser ce travail.

Une fois les langages choisis, nous nous sommes renseignés sur le fonctionnement des ontologies. Réaliser une ontologie demande à la fois de comprendre ses divers mécanismes, sa sémantique et de savoir utiliser les outils permettant de la manipuler. La première chose à faire est donc de se renseigner sur les concepts utilisés par les ontologies comme les logiques de description qui reposent sur la logique du premier ordre.

Après cette période de recherche vient la création d'un état de l'art sur l'utilisation des ontologies, et plus particulièrement celles appliquées aux langages de programmation. Cela a permis de mettre en avant les ontologies déjà existantes, la manière dont elles ont été construites et leurs limitations.

Ensuite, il a fallu se renseigner sur la manière dont l'héritage fonctionne pour chaque langage de programmation. Pour ce faire, il a fallu parcourir la documentation officielle des divers langages pour dresser un tableau récapitulatif avec les principaux concepts de chaque langage.

En se basant sur les analyses réalisées et avant de se lancer la création des ontologies, un méta-modèle a été réalisé. Ce dernier permet d'avoir une vision plus claire pour les personnes qui ne sont pas familières avec les ontologies mais également d'apporter de la précision sur les différentes notions qu'il faudra manipuler. Ce fut un travail itératif où le méta-modèle a été modifié à de nombreuses reprises afin d'être le plus juste possible.

Enfin, vient la création des ontologies sur base du schéma réalisé précédemment. Les ontologies ont été conçues en respectant au maximum les conventions généralement adoptées et en se basant sur le tutoriel fourni par l'université de Stanford pour prendre en main les ontologies et l'outil développé par cette-même université (Protégé [2]). Tout comme la conception du méta-modèle, la création d'une ontologie est un processus itératif qui demande d'être refait en fonction des précisions et corrections à apporter.

1. https ://pypl.github.io/PYPL.html
2. https ://protege.stanford.edu/

1.4 Organisation du mémoire

Le mémoire est découpé en plusieurs parties, chacune traitant un des points explicités dans la section sur la méthodologie. De ce fait, le mémoire commence par un chapitre sur les prérequis, à savoir la logique du premier ordre, les logiques de description et les ontologies. Ensuite, le chapitre dédié à l'état de l'art reprend ce qui existe dans la littérature. Le chapitre suivant concerne l'analyse de l'héritage dans les divers langages étudiés. Les deux parties suivantes présentent respectivement la création du méta-modèle et celle de l'ontologie proposée. Le mémoire se conclut par une partie concernant les limites de ce travail ainsi que les éventuelles améliorations que l'on pourrait apporter concernant de futurs travaux. Enfin, la conclusion récapitule ce qui ressort de ce mémoire.

2 Prérequis

Avant de s'intéresser plus en avant sur les ontologies en elles-mêmes, il est important d'expliquer, brièvement, les principes qui servent de base à la plupart des ontologies. Les ontologies reposent sur les logiques de description qui permettent de représenter un ensemble de connaissances sur un domaine en particulier. Ces connaissances sont exprimées sous forme de prédicats [3].

2.1 Histoire des logiques de description

Les logiques de description sont le prolongement d'un travail de recherche entamé durant les années 1970 [4] où le but était de pouvoir construire des systèmes capables d'inférer des faits à partir d'une base de connaissances. Selon Nardi et Brachman, il y avait à cette époque deux types d'approches : une approche basée sur le calcul des prédicats et une autre plus basée sur la cognition que sur la logique. La première approche usait donc d'une formalisation logique de la connaissance tandis que la seconde s'aidait d'interfaces graphiques évocatrices [43].

L'approche logique était utilisée afin de fournir des résultats pouvant servir dans des cas génériques tandis que l'approche cognitive était pratiquée pour des problèmes spécifiques, même si les résultats de cette dernière pouvaient être généralisés. La seconde approche était souvent regardée comme étant l'approche la plus populaire étant donné sa facilité de compréhension par rapport à l'approche logique, plus complexe à comprendre et à maîtriser.

Un exemple typique de l'approche logique serait le General Problem Solver (GPS) de 1959 par Newell et al. qui, sur base d'heuristiques [5] et de problèmes énoncés dans un certain formalisme (comme des clauses de Horn), permettait d'arriver à une solution. Ce genre de système ne fonctionnait pas avec des problèmes trop spécifiques et avait plusieurs limitations comme la complexité d'énoncer les problèmes dans un formalisme supporté [45].

Il existe plusieurs exemples d'avancée dans l'approche cognitive. On peut citer les travaux de Quillian qui servirent de base aux réseaux sémantiques [6]. L'objectif de ces travaux était de présenter un modèle mettant en relation différents concepts, en partant de mots définis dans un dictionnaire. Il souhaitait ainsi imiter la mémoire à long terme de l'être humain et l'appliquer à l'informatique [51]. Un autre exemple possible serait les travaux de Minsky sur sa représentation de la connaissance sous forme de cadres (frames). Ces travaux permettent de représenter une situation bien stéréotypée et d'avoir un cadre qui contient diverses informations sur cette situation en particulier. Parmi ces informations on retrouve des explications sur la situation, sur ce qu'il va se produire ensuite et sur les actions à entreprendre [41].

Un des avantages principaux de ce genre de représentation de la connaissance est le fait que l'être humain puisse facilement comprendre les informations à sa disposition grâce à une interface visuelle claire. Mais la contrepartie est que ces approches manquent d'un formalisme qui rend difficile la ré-exploitation et la combinaison de plusieurs modèles de connaissance [43].

Les logiques de description vont donc essayer de mélanger ces deux aspects, logique et cognitif,

3. Les prédicats correspondent à des notations mathématiques exprimant des propriétés du domaine étudié.
4. Certains travaux datent des années 1950 mais c'est dans les années 1970 que l'intérêt grandit
5. On parle ici d'informations incomplètes ou imprécises qui peuvent aider à la résolution d'un problème.
6. Représentation de plusieurs concepts liés entre eux par une relation.

afin d'avoir la sémantique formelle fournie par les calculs de prédicats ainsi que la représentation visuelle (entre autres) provenant des systèmes de cadres et des réseaux sémantiques [21].

Pour établir les divers prédicats nécessaires, les logiques de description utilisent ce qu'on appelle la logique du premier ordre, synonyme de calcul de prédicats.

2.2 La logique du premier ordre

L'objectif de cette sous-section n'est pas d'expliquer en profondeur le fonctionnement de la logique du premier ordre ni d'étudier son histoire mais simplement d'expliquer brièvement ce que c'est et comment cela est utilisé dans les logiques de description.

La logique du premier ordre permet d'exprimer de manière formelle les propriétés d'objets et les relations entre plusieurs de ces objets. Bien que les logiques de description n'utilisent que la logique du premier ordre, il existe des logiques d'ordre supérieur, telles que la logique du second ordre ou du troisième ordre. Ces dernières permettent de traiter, entre autres, les fonctions comme étant de simples variables, notamment en leur attribuant des quantificateurs. Elles peuvent aussi utiliser une sémantique plus rigoureuse que la logique du premier ordre [53].

Afin de pouvoir répondre aux attentes de la logique du premier ordre, une grammaire a été développée et est composée plusieurs symboles. Barwise J. présente dans [22] ceux qui seront le plus utilisés (la liste est donc non exhaustive) :

- des connecteurs logiques : $\wedge$ (et), $\vee$ (ou), $\neg$ (négation), $\rightarrow$ (implique), ...
- des symboles de comparaison : $=$, $<$, $>$, ...
- des quantificateurs : $\forall$ (universel) et $\exists$ (existentiel)
- des prédicats qui expriment des relations : estUnHomme(x), ...
- des symboles opérant sur les groupements : $\in$ (inclusion), $\subset$ (sous-groupement), $\cap$ (intersection), $\cup$ (union), ...
- des variables : a, b, c, ...
- des fonctions : f, g, ...
- des constantes : 10, 'Mary', ...
- des symboles permettant de grouper et d'ajouter de la lisibilité : (et)
- d'autres symboles encore qui dépendent du domaine étudié

A l'aide de ces symboles, qui forment en fait le vocabulaire de la logique du premier ordre, il est possible d'exprimer plusieurs faits et règles de manière formelle.

La logique du premier ordre permet de représenter une phrase telle que "Tous les hommes sont mortels, Socrate est un homme donc Socrate est mortel" de la manière suivante :

$$\forall x\ \text{homme}(x) \rightarrow \text{mortel}(x)$$
$$\text{homme('Socrate')}$$

La dernière partie de la phrase pourra être inférée grâce aux prédicats fournis. En effet, on peut déduire de par l'implication et de par le fait que Socrate soit un homme, que Socrate est bel et bien mortel :

$$\text{mortel('Socrate')}$$

2.3 Fonctionnement des logiques de description

Les logiques de description emploient plusieurs notions, à savoir les concepts, les individus (ou instances), les rôles, les attributs et les axiomes. Les rôles et les attributs forment ce qu'on appelle communément les propriétés.

1. Les concepts : ils représentent les concepts du domaine décrit. Ils peuvent posséder diverses propriétés qui ont pour but d'enrichir les concepts en y apportant un niveau de détail plus élevé. Ces propriétés possèdent elles-mêmes des caractéristiques permettant d'améliorer l'inférence. Il est alors possible de préciser si une propriété est l'inverse d'une autre ou, dans le cas d'un rôle, s'il est "Functional" (ce qui implique que cette propriété n'autorise qu'une relation avec au plus une autre instance).
2. Les instances : ce sont généralement des objets concrets appartenant à un certain concept.
3. Les rôles : ils permettent à lier les objets entre eux. Ils peuvent avoir des contraintes, comme des cardinalités minimales ou maximales à respecter [7]. Les rôles sont autant utiles pour définir des liens entre concepts qu'entre instances de ces concepts. On pourrait dire que le rôle est lui-même instancié.
4. Les attributs : Ils représentent des données propres à un individu. Ils peuvent être de plusieurs types (une chaine de caractères, un entier, ...).
5. Les axiomes : ils permettent d'ajouter des contraintes supplémentaires, telles une quantité maximum ou une plage de valeurs acceptées.

Pour illustrer ces notions, si on s'intéresse aux animaux, on pourrait dire que les mammifères représentent un concept. Un individu du concept mammifère pourrait donc être un chien nommé "Médor". Un attribut pourrait être l'âge du mammifère (en année, en mois, peu importe), dans ce cas-ci, l'âge du chien. On pourrait définir un axiome disant qu'un jeune chien possède un âge inférieur à 3 mois. Un rôle possible serait celui qui lie le chien à ses parents (donc à d'autres individus).

Comme expliqué par [43], ces entités peuvent être liées les unes aux autres. Cela peut se faire de deux manières différentes :

— par un rôle qui définit une relation, symbolisée par un prédicat binaire

— par une relation de subsomption : On spécifie qu'un concept est subsumé par un autre concept. Cette relation permet d'indiquer si un concept peut être inclus dans un autre. Cela peut également se traduire par une relation d'implication, où $A \rightarrow B$ est semblable à A est subsumé par B (ou encore $A \sqsubseteq B$). Grâce à cette relation, le concept subsumé obtient les propriétés définies dans le concept parent ainsi que les potentiels rôles dudit parent. En règle générale, les concepts seront représentés par des prédicats unaires.

Les connecteurs logiques de la logique du premier ordre sont représentés en utilisant les symboles opérant sur les groupements (union, intersection...). Concrètement, pour exprimer en logique du premier ordre le $\wedge$ ("et"), le symbole $\sqcap$ sera utilisé. Par exemple, pour lier deux concepts, on aurait en logique du premier ordre :

$$\text{Concept1(x)} \wedge \text{Concept2(x)}$$

Le résultat en logiques de description :

7. Les cardinalités indiquent le nombre d'instance qui peut être concerné par un rôle.

$$\text{Concept1} \sqcap \text{Concept2}$$

De la même manière, le $\vee$ ("ou") deviendra $\sqcup$ (union).

Les concepts peuvent eux-mêmes être définis à partir d'une combinaison de concepts et de rôles, liés entre eux par des connecteurs logiques.

Par exemple, le concept représentant un père pourrait être défini comme étant le résultat de la combinaison entre le concept "Homme" et le rôle "aUnEnfant", ce qui se traduit en logiques de description par :

$$\text{Pere} : \text{Homme} \sqcap \text{aUnEnfant}$$

Sur ces concepts, il est possible d'effectuer plusieurs opérations de raisonnement telles que vérifier la satisfiabilité d'un concept ou déterminer si la relation de subsomption entre deux concepts est valide. Un concept est satisfiable s'il est possible de trouver des individus le composant, en d'autres termes, qu'il puisse être défini par autre chose que l'ensemble vide. Toutes ces opérations sont effectuées par des raisonneurs qui utilisent des algorithmes qui ont été créés et peaufinés depuis les années 1970, comme par exemple [23]. Daniele Nardi, Ronald J Brachman, et al., dans [43], expliquent plus en détails les différentes relations d'inférence qui peuvent être appliquées aux concepts et résument les évolutions dans les techniques de raisonnement.

En logique de description, il est possible de décrire les concepts (et les rôles) de deux manières différentes. Soit on les décrit d'un point de vue global, décrivant des généralités sur le domaine étudié soit on peut s'intéresser à des individus représentant les concepts en question et étudier les relations (rôles) qui les lient. Ces deux manières correspondent, respectivement, à ce qu'on appelle la TBox (T pour Terminologie) et la ABox (A pour assertion) [26]. Pour illustrer ces deux concepts, on pourrait repartir sur l'exemple du concept de Père. Pour la TBox, on obtient ceci :

$$\text{Pere} : \text{Homme} \sqcap \text{aUnEnfant}$$

Et pour la ABox, si on désire parler d'un individu qu'on appellera Thierry et qui possède un enfant nommé Henry :

$$\text{Homme(THIERRY)} \sqcap \text{aUnEnfant(THIERRY, HENRY)}$$

Ces deux notions représentent donc le coeur des logiques de description. En effet, toujours selon [26], les logiques de description peuvent être découpées en 4 grandes idées, deux d'entre elles étant la TBox et la ABox. Les concepts restants sont :

1. L'ensemble des expressions permettant de définir les concepts et rôles du domaine étudié.
2. Les inférences qu'il est possible d'effectuer sur les éléments appartenant à la TBox et à la ABox.

Un dernier point qu'il est important de souligner dans le fonctionnement des logiques de description est que le domaine étudié peut être fini ou non fini. Dans le cas d'un domaine non fini, on parlera alors de "open-world assumption" (OWA), indiquant ainsi qu'il est impossible de tout savoir et que les inférences qui seraient effectuées par un raisonneur pourraient être limitées par rapport à ce qu'elles donneraient dans un domaine fini (ou "closed-world assumption") [43].

Maintenant que les notions de base des logiques de description ont été présentées, on peut alors s'intéresser aux ontologies.

2.4 Les ontologies

Cette section a pour vocation de présenter les ontologies en partant de leur définition première pour arriver à la définition qu'elles ont aujourd'hui. Quelques types d'ontologies sont également présentés avant d'arriver à l'exposition des différents langages utilisés pour construire des ontologies.

2.4.1 Définitions

L'ontologie au sens premier du terme provient de la métaphysique, la partie de la philosophie qui s'intéresse à l'étude de l'être. On retrouve les premières références au mot "ontologie" aux alentours de 1613 par deux philosophes [54]. Le dictionnaire Le Robert la définit comme telle : "Partie de la métaphysique qui traite de l'être indépendamment de ses déterminations particulières". Le Robert propose également une seconde définition, axée sur l'informatique : "Ensemble structuré de concepts permettant de donner un sens aux informations" [3].

D'un point de vue informatique, les chercheurs et professionnels commencèrent à s'intéresser aux ontologies et à ce qu'elles pouvaient apporter dès les années 1970, voire depuis les années 1960 puisque à cette époque des programmes servant de base de connaissance commençaient à voir le jour [42]. Le milieu de l'intelligence artificielle est celui qui s'intéressa le plus aux ontologies et aux modèles de calculs automatisables en général. En 1980, McCarthy publie un papier où il explique comment il serait possible de raisonner automatiquement en utilisant la logique du premier ordre [39]. Barry Smith and Christopher Welty proposent une étude plus précise de l'histoire de l'ontologie et discutent des domaines dans lesquels les ontologies pourraient s'avérer pertinentes [54]. Ce travail met d'ailleurs en avant les différentes définitions qui ont été attribuées au terme "ontologie" en fonction des domaines où il était utilisé.

En 1991, les ontologies sont vues comme étant des outils permettant de partager et de réutiliser des bases de connaissance, toujours principalement dans le cadre de l'intelligence artificielle. C'est durant cette année que Neches et al. proposeront alors une définition : « Une ontologie définit les termes et relations basiques qui composent le vocabulaire d'un domaine ainsi que les règles pour combiner ces termes et relations afin de définir des extensions au vocabulaire »[44].

Deux ans plus tard, en 1993, Gruber propose une nouvelle définition qui deviendra la plus utilisée. Il définit l'ontologie comme étant une « spécification explicite d'une conceptualisation »[33] . Cette définition sert de base à d'autres définitions, plus affinées ou adaptées au contexte des ontologies représentées [31]. Telle la version retravaillée par Borst en 1997 qui a modifié la définition de Gruber en : « Une ontologie est une spécification formelle d'une conceptualisation partagée »[24].

Les ontologies permettent, entre autres, de :

- représenter de manière formelle les connaissances sur le domaine étudié,
- permettre et favoriser la réutilisation des ontologies existantes,
- définir clairement les concepts étudiés et instaurer des conventions de nommage.

La construction d'une ontologie change en fonction du domaine d'intérêt, des potentielles conventions internes et, principalement, de l'expérience des personnes réalisant l'ontologie (que ce soient des chercheurs, des analystes, ...). Néanmoins, il existe plusieurs guides permettant d'aider à la création d'ontologies en suivant une certaine méthodologie. Parmi ces guides, on retrouve notamment celui de Noy et Mcguinness [46] qui, bien que de datant de 2001, reste d'actualité et expose les enjeux et difficultés relatifs à la construction d'ontologies. Ce guide indique bien qu'il n'existe

aucune solution miracle pour rédiger une ontologie et qu'il est possible d'en rédiger de plusieurs manières. Noy et Mcguinness ont établi une méthodologie en 7 étapes qui permet d'aider grandement à la réalisation d'une ontologie. Ces étapes sont les suivantes [8] :

1. « Déterminer le domaine et le périmètre de l'ontologie »
2. « Envisager de réutiliser des ontologies existantes »
3. « Énumérer les termes importants dans l'ontologie »
4. « Définir les classes et la hiérarchie de classes »
5. « Définir les propriétés des classes »
6. « Définir les aspects des propriétés »
7. « Créer des instances »

Les ontologies reposent en partie sur les logiques de description et partagent donc plusieurs notions communes. Ces dernières comprennent les concepts (qui deviennent des classes), les instances (des individus), les rôles, les attributs et les axiomes. En fonction du langage choisi pour construire l'ontologie, ces notions peuvent varier et être plus ou moins enrichies.

2.4.2 Types d'ontologies

Outre le fait de retracer les différentes définitions modernes d'une ontologie, [31] reprend également plusieurs types d'ontologies possibles tout en fournissant des exemples connus afin d'illustrer ces différents types. Chaque type d'ontologie poursuit un but différent. Parmi ces types, on peut citer l'ontologie linguistique qui a pour objectif principal de décrire les langues et le vocabulaire les composant (comme par exemple WordNet [9]). D'après [57], ce serait le type d'ontologie le plus répandu. Il existe aussi la version meta-linguistique qui, comme son nom l'indique, s'intéresse plutôt aux concepts propres aux langues, tels que la grammaire, la conjugaison ou la manière dont les phrases sont constituées.

Un autre genre d'ontologie souvent utilisé est l'ontologie de représentation des connaissances. Cette dernière permet de décrire un domaine et les connaissances qui lui sont attachées. Un exemple d'une telle ontologie est la « Frame Ontology »présentée par Gruber, qui est un composant principal de l'outil « Ontolingua » [10], outil permettant de traduire des ontologies vers différents systèmes de représentation (LOOM, Epikit, ...). Pour ce faire, la « Frame Ontology »nécessite un ensemble de définitions composant l'ontologie de base. Ces définitions seront transformées en propositions écrites en respectant la notation imposée par le langage KIF (Knowledge Interchange Format), langage ayant pour but de partager la connaissance sans pour autant permettre des actions telles que l'inférence [33].

En addition aux deux types mentionnés précédemment, [31] présente également les « ontologies de tâche »et les « ontologies de domaines-tâche ». Les ontologies appartenant à ces catégories ont pour objectif d'aider à la résolution de certains types de problèmes. Dans le cas des « ontologies de domaines-tâche », les problèmes appartiennent à un même domaine alors que pour les « ontologies de tâche », le domaine des problèmes importe peu.

8. Traduit depuis l'anglais
9. https ://wordnet.princeton.edu/
10. http ://www.ksl.stanford.edu/software/ontolingua/

2.4.3 Différents langages

Puisque cela fait maintenant plusieurs dizaines d'années que les ontologies existent, elles ont connu de nombreux changements, tant dans la façon dont elles sont construites que dans la manière dont elles sont utilisées.

Parmi ces changements, un des plus importants est l'évolution des langages utilisés pour constituer les ontologies. Ces langages ont pour certains évolué, tandis que d'autres ont disparu car ils n'étaient plus à jour avec ce qui se faisait ou étaient trop limitatifs. Bien que ce travail se fasse en utilisant le langage OWL, il est pertinent de brièvement parler des autres langages encore utilisés de nos jours.

D'après l'enquête réalisée par Kalibatiene et Vasilecas publiée en 2011 [36], 4 langages ressortent par rapport aux autres : KIF, RDF + RDF(S), DAML+OIL et OWL. Ces langages sont succinctement décrits ci-après.

Le langage KIF, déjà brièvement défini précédemment, a donc pour but de permettre le transfert d'information entre différents programmes. Il fut mis au point par un organisme appelé la « Knowledge Sharing Effort », financé par plusieurs organisations dont la DARPA (Defense Advanced Research Projects Agency), afin de répondre à la problématique qui se posait à l'époque de pouvoir partager la connaissance entre plusieurs systèmes ou d'exploiter des ressources existantes [48]. Il est aujourd'hui décliné en plusieurs versions.

Le langage RDF + RDF(S) est le résultat du langage RDF (Resource Description Framework) et de RDF Schéma. Le W3C (World Wide Web Consortium) a conçu RDF afin de faciliter l'usage des ressources se trouvant sur le web. En effet, ce langage, devenu un standard, permet aux ordinateurs d'échanger et de communiquer à propos de ces ressources en les identifiant grâce à leur URI (Uniform Resource Identifier). RDF(S) est une extension au langage RDF améliorant l'expressivité et apportant une plus grande souplesse aux développeurs [36].

DAML+OIL est composé de deux langages, DAML (DARPA Agent Markup Language) et OIL (Ontology Interchange Language), qui ont comme points communs d'avoir été développés pour supporter les ontologies web ainsi que d'être compatibles avec XML et RDF, ce qui leur permet d'exploiter le web sémantique. Le langage OIL, basé sur RDF(S), a été créé à la suite de demandes émises par un groupe composé principalement de chercheurs européens, toujours selon [36]. Le langage DAML est en quelque sorte la contrepartie américaine de OIL. Il offre les mêmes possibilités et possède son propre langage permettant de gérer des contraintes et des inférences.

Le langage OWL 1.0 est lui aussi un standard du web sémantique conçu par le W3C en 2004 [11]. OWL se base également sur le langage RDF(S) et est souvent divisé en 3 parties : OWL Lite, OWL DL et OWL Full. OWL a été découpé ainsi car il s'agit d'un langage complet ce qui a pour effet de parfois apporter de la lourdeur et de la complexité quand cela n'est pas nécessaire. OWL Lite a pour avantage qu'il est simple et léger mais ceci a pour conséquence que son expressivité s'en voit réduite. OWL DL (pour Descriptive Logics) permet d'être plus expressif et d'utiliser les moteurs d'inférence à leur plein potentiel, mais cela se fait au coût de la compatibilité avec RDF. OWL Full permet une expressivité maximale tout en étant totalement compatible avec RDF. Mais en contrepartie, l'utilisation de moteurs d'inférence dans OWL Full devient compliquée, voire impossible [32]. OWL Full est le sous-langage correspondant le plus au langage RDF, ce qui fait qu'un document utilisant

11. De nos jours, lorsque l'on fait référence à OWL, il s'agit souvent de sa version 2.0 sorti en 2012.

RDF peut être traduit facilement en OWL Full. Ce ne sera pas nécessairement le cas avec OWL Lite et OWL DL [59].

La Figure 1 reprend de manière schématique les trois sous-langages composant OWL. Cette image a été réalisée à partir de celle se trouvant dans un document présenté lors du W3C Day de la conférence Evolve en décembre 2004 à Brisbane [52].

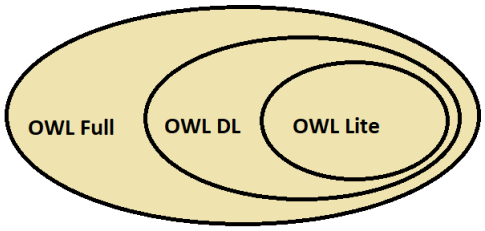

FIGURE 1 – Les différentes parties du langage OWL

Chacun de ces langages possède ses forces et ses faiblesses. L'enquête de Kalibatiene et Vasilecas les analyse et les compare sur base de 13 critères déterminés après consultation de plusieurs travaux effectués sur ces langages. Ces critères sont cités ci-dessous :

1. La construction ou conceptualisation d'un domaine
2. Spécifications des perspectives
3. L'expressivité
4. La compréhensibilité
5. La formalité
6. Le moteur d'inférence
7. La vérification des contraintes
8. L'implémentation (dans un outil)
9. La correspondance avec d'autres langages
10. Le paradigme [12] utilisé
11. Les standards web utilisés
12. Le langage est-il considéré comme un standard
13. La popularité

Il ressort de cette enquête plusieurs résultats intéressants, notamment en ce qui concerne la popularité des langages. Voici un extrait de leur tableau récapitulatif :

Critère	KIF	OWL	RDF + RDF(S)	DAML+OIL
Popularité	29 100	429 000	137 000	10 500

La popularité correspond en fait au nombre de liens Google Scholar. Bien que cela ne reflète pas exactement la popularité d'un langage ou son adoption par la communauté, cela permet toutefois d'avoir une bonne idée sur son utilisation. On constate alors qu'OWL est bien plus populaire que

12. Un paradigme correspond à ensemble de concepts utilisés pour répondre à un certain type de problématique.

ses homologues. Cette différence peut s'expliquer par le fait qu'OWL (et à fortiori OWL 2.0 [13]) est un langage plus jeune que les autres et activement mis à jour. DAML+OIL date de 2001, KIF d'avant 1992 (date de la draft de la version 3.0) et RDF de 1999. Outre sa jeunesse, le fait qu'il soit considéré comme un standard par le W3C lorsqu'il s'agit des ontologies web favorise également son utilisation auprès de la communauté.

Puisque ce mémoire se fait en utilisant OWL, et plus particulièrement OWL DL, nous allons un peu plus le détailler. Le langage OWL possède des similitudes avec le paradigme orienté objet dans sa manière de découper les éléments composant une ontologie. En effet, OWL utilise des notions de classes qui peuvent avoir des propriétés et où chaque classe peut être représentée par une ou plusieurs instances. Ces instances possèdent des valeurs concrètes pour les propriétés définies précédemment. Tout comme dans les logiques de description, il est possible d'avoir de l'héritage entre les classes, toujours de manière semblable au paradigme orienté objet. En fait, l'héritage entre les classes est même une composante essentielle du langage OWL car cela aide, entre autres, à inférer des résultats. OWL DL permet d'avoir la même expressivité que OWL Lite avec pour différence principale que les cardinalités des contraintes ne sont pas limitées à 0 ou 1, comme c'est le cas pour la version Lite [38]. OWL autorise d'avoir des classes nommées et d'autres qui sont dites anonymes. Ces dernières sont généralement le résultat d'une construction de classes nommées, anonymes ou d'axiomes, et ce peu importe le ou les opérateurs utilisés. Les classes peuvent être disjointes les unes des autres ou au contraire équivalentes. OWL permet de renseigner une série de caractéristiques sur les propriétés qui composent l'ontologie. Les rôles, qui permettent donc de faire le lien entre plusieurs classes, possèdent quelques caractéristiques que les simples attributs n'ont pas. On dénombre entre autres la possibilité d'indiquer si un rôle est "Functional" ou non, s'il est transitif, (ir)réfléchi ou (a)symétrique. Ces propriétés sont tirées depuis l'outil Protégé [14], un des outils disponibles pour la création d'ontologie en OWL. Outre cela, pour chaque propriété, on peut aussi renseigner une propriété inverse, indiquer s'il s'agit d'une sous-propriété et spécifier les types qu'elle peut prendre. Tout comme les logiques de description, OWL respecte le principe d'OWA (Open-world assumption), notion importante à savoir lorsque l'on construit une ontologie puisque ce qui n'est pas explicitement interdit est considéré comme autorisé.

2.4.4 Requêtage et raisonneur

Une fois l'ontologie constituée et prête à l'emploi, il faut encore pouvoir l'exploiter. Typiquement, on va utiliser un langage de requête permettant d'obtenir des informations à partir de l'ontologie. Il existe plusieurs langages de requêtes, comme par exemple SPARQL [15] ou OWL-QL [16]. OWL-QL permet de requêter une (ou plusieurs) base de connaissance liée au web sémantique grâce à un système de "requête-dialogue". Il a été prévu pour fonctionner avec OWL mais, de par sa conception, peut être simplement converti pour travailler avec d'autre langages reposant sur la logique du premier ordre (tel que KIF) [30]. SPARQL quant à lui, est spécialisé pour travailler avec les ontologies reposant sur RDF [17].

13. https ://www.w3.org/TR/owl2-overview/
14. http ://protegeproject.github.io/protege/views/object-property-characteristics
15. https ://www.w3.org/TR/sparql11-overview/
16. https ://www.w3.org/TR/owl2-profiles/$\mathrm{OWL}_{2Q}L$
17. Bien qu'il existe une extension permettant d'utiliser des ontologies OWL directement : https ://www.w3.org/2009/sparql/wiki/Feature :SPARQL/OWL

Les raisonneurs ont pour objectif d'inférer ce qui peut l'être à partir d'une certaine ontologie. Il en existe plusieurs, tous ayant leurs spécificités. Certains conviennent mieux pour les ontologies plus chargées et précises, d'autres excellent pour les ontologies plus petites. L'outil Protégé propose une série de raisonneurs par défaut tels que Hermit [18] et Pellet [19]. Une des utilisations possibles d'un raisonneur est de s'assurer que l'ontologie est cohérente et ne dispose donc pas de contradiction. Le cas échéant, le raisonneur indique où le problème a été détecté et ce qui pourrait en être la cause.

Voilà ce qui conclut la présentation des ontologies, nous allons à présent passer à l'état de l'art en soi.

18. http ://www.hermit-reasoner.com/
19. https ://github.com/stardog-union/pellet

3 État de l'art

Maintenant que les bases ont été posées, nous allons pouvoir commencer l'état de l'art. Dans cette section, nous allons faire le tour de ce qui existe déjà dans la littérature vis-à-vis des ontologies liées aux langages de programmation, et plus particulièrement à celles liées au concept d'héritage, s'il y en a.

3.1 L'enseignement des langages de programmation

Les ontologies sont généralement développées afin de répondre à un ou plusieurs besoins. Dans le cadre des langages de programmation, il se trouve que la thématique qui ressort le plus souvent de la recherche menée est celle de l'enseignement.

Les professeurs se servent des ontologies afin de faciliter l'apprentissage des langages de programmation et de leurs concepts auprès de leurs étudiants. En effet, comme expliqué par Pierrakeas, Solomou et Kameas dans [49] les ontologies sont particulièrement pertinentes lorsqu'il s'agit d'enseigner puisqu'elles sont facilement réutilisables et constituent un bon support de cours, définissant de manière formelle les termes employés. Dans leur article, les auteurs emploient des "objets d'enseignement-apprentissage" [20] qui représentent une matière à donner divisée en plus petits blocs logiques [58] et qui se basent sur un système de mots-clés. [49] propose alors de synchroniser les mots-clés utilisés dans les objets d'enseignement-apprentissage avec les termes utilisés dans l'ontologie qui représente le domaine étudié, dans ce cas un langage de programmation. Dans cette étude, deux ontologies sont créées, une pour le langage Java et une pour le langage C, toutes les deux en suivant la méthodologie présentée par Noy et Mcguinness [46]. Bien que ces ontologies ne soient pas spécialement en lien avec l'héritage (à fortiori pour le langage C qui ne supporte pas l'héritage), elles offrent déjà un aperçu des différents concepts présents dans un langage de programmation. Il ressort de ce papier que l'utilisation d'ontologies couplées avec des objets d'enseignement-apprentissage permet aux enseignants d'avoir une meilleure vue sur leur matière ainsi que d'identifier des chemins logiques d'apprentissage en fonction des relations liant les concepts dans l'ontologie.

[37] proposé par Kouneli et al. est la continuité de l'étude citée précédemment. Dans ce document, une nouvelle ontologie concernant le langage Java est créée, toujours en suivant la méthodologie proposée par Noy et Mcguinness. Kouneli et al. se sont basés sur la documentation officielle de Java, proposée par Oracle [21], afin de recenser tous les concepts que l'ontologie devra être capable de supporter. Cette ontologie est une version améliorée de celle proposée précédemment car elle corrige certains des problèmes identifiés, comme l'utilisation d'une chaîne de caractères au lieu d'un rôle. Elle apporte également des nuances qui permettent de rendre l'ontologie plus efficace. Il est à noter que cette ontologie et celles construites dans [49] ont été développées dans le langage OWL et en utilisant l'outil Protégé.

Un autre cas d'ontologie utilisée dans le cadre scolaire est l'ontologie développée par Sosnovsky et Gavrilova [55] qui représente elle aussi le langage C. Les buts poursuivis par cette ontologie sont les mêmes que ceux explicités précédemment, à savoir l'optimisation de la manière d'enseigner le cours et la réutilisabilité des supports d'enseignement. En outre, cela permettrait aux programmes déjà mis en place au sein de l'université de dialoguer avec cette ontologie et, dans le futur, de

20. Appelés "Learning objects" en anglais.
21. http ://docs.oracle.com/javase/tutorial/

permettre à des applications tierces de se greffer à cette ontologie. Contrairement aux deux travaux mentionnés ci-avant, Sosnovsky et Gavrilova proposent leur propre algorithme permettant de construire une ontologie. Même si l'ontologie produite étudie le même domaine que [49], elle est néanmoins différente. D'une part, l'article ne précise pas dans quel langage l'ontologie est spécifiée et, d'autre part, seul un modèle représentant les concepts à inclure dans l'ontologie est présenté. Donc, bien que ce papier présente une méthode légèrement différente de celle démontrée par Noy et Mcguinness, son apport quant à l'utilisation d'ontologies pour représenter les langages de programmation reste assez sommaire.

[28] écrit par Epure et Iftene porte sur la représentation du langage C#. Les auteurs ont décidé de partir d'une ontologie existante sur le langage Java, écrite en OWL et de l'adapter afin qu'elle corresponde aux spécificités du langage C#. L'ontologie de départ, développée par Alnusair et Zhao [19] et appelée SCRO (Source Code Representation Ontology), a pour fonction de faciliter la réutilisabilité de différents composants. Pour y arriver, le squelette de l'ontologie est spécifié à la main (les concepts, les rôles, les axiomes, ...) et les instances des concepts renseignés seront générées grâce à plusieurs artefacts (code source) se trouvant dans un certain dépôt. Pour exploiter ces artefacts, un extracteur spécialisé pour le langage Java a été développé. Outre l'adaptation de l'ontologie SCRO, [28] propose également un système permettant de générer des triplets RDF à partir d'un code source C# et de l'ontologie nouvellement créée.

Une dernière ontologie créée dans le but d'enseigner qui est intéressante à mentionner ici est celle mise au point par Diatta, Basse et Ouya [27]. Le langage choisi est le langage Pascal car, d'après les recherches qu'ils ont menées, les ontologies déjà existantes se concentraient principalement sur les langages Java et C, or ils considèrent le langage Pascal comme étant particulièrement adapté pour une introduction à la programmation. L'ontologie proposée repose également sur le langage OWL et sa construction suit les préceptes proposés par Noy et Mcguinness. Elle reprend les différents concepts composant le langage Pascal. L'ontologie permet de simplifier la gestion des exercices donnés aux étudiants et une interface homme-machine permettant de requêter l'ontologie (via SPARQL) a été développée.

3.2 Les langages de programmation orienté objet

Le chapitre précédent a mis en avant des ontologies créées pour satisfaire un seul langage. Or, nous souhaitons une vision plus large, une ontologie permettant de traiter plusieurs langages à la fois, tant qu'ils ont des points communs les permettant d'être comparés, comme le fait d'exploiter le paradigme orienté objet.

Parmi les études existantes, Abuhassan et Almashaykhi présentent une ontologie combinant plusieurs langages orienté objet, entre autres Java et C++ [18]. Les auteurs ont opté pour une approche via des schémas UML afin de modéliser le domaine de connaissance à représenter et de délimiter les concepts à intégrer dans l'ontologie. Bien que ce papier constitue une bonne base dans la représentation de plusieurs langages au sein d'une seule ontologie, la partie dédiée à la création de l'ontologie est assez succincte. L'ontologie, écrite en OWL, ne suit pas une méthodologie bien définie. Cela n'aide pas à la compréhension et rend l'ontologie difficilement exploitable.

Atzeni et Atzori proposent dans [20] une ontologie (en OWL) qui regroupe les grands concepts propres aux langages orienté objet, bien qu'ils se soient principalement inspirés du langage Java. Ils assurent que la conception de l'ontologie a été faite de manière à pouvoir aisément être modifiée

afin de correspondre à d'autre langages de programmation. Ils indiquent que leur ontologie est similaire à celle présentée dans [19], avec quelques différences. En effet, la version de Atzeni et Atzori est plus récente et permet de gérer des cas qui ne l'étaient pas auparavant, comme les types génériques. Un petit peu comme [28], les auteurs proposent leur propre parser [22] de code source, basé sur le langage Java, qui permet de générer des triplets RDF, qui peuvent à leur tour être utilisés via des requêtes rédigées en SPARQL. Ils proposent une utilisation maximale des données fournies par le web sémantique. Ils ont publié leur ontologie sur le web, la rendant ainsi accessible à qui veut (à condition de respecter la licence CC BY 4.0). En ouvrant cette ontologie (via Protégé par exemple), on se rend compte que l'ontologie ne supporte que l'héritage simple et non l'héritage multiple. Cela est logique puisqu'elle est basée sur le langage Java mais est fortement restrictif pour d'autres langages orienté objet, tel que C++, qui autorisent l'héritage multiple.

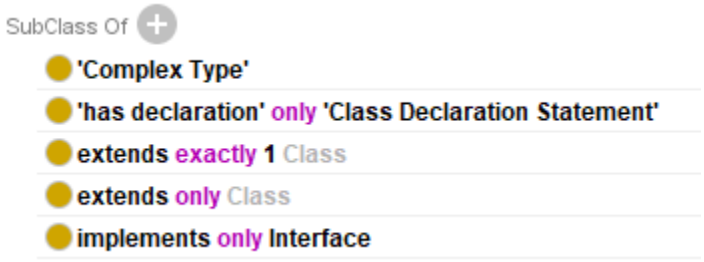

FIGURE 2 – Exemple de l'héritage simple

Enfin, l'article le plus pertinent pour notre recherche provient du livre relatant la 38ème conférence sur la modélisation conceptuelle qui a eu lieu au Salvador et au Brésil en 2019. Le chapitre, écrit par de Aguiar, de Almeida Falbo et Souza, intitulé « A Reference Ontology on Object-Oriented Code »[25], est celui qui nous intéresse. Dans ce chapitre, les auteurs expliquent n'avoir trouvé aucun consensus sur la manière dont les langages de programmation orienté objet sont modélisés. C'est pour cette raison qu'ils vont proposer leur propre ontologie qui va permettre de représenter les concepts communs aux langages orienté objet. Pour développer leur ontologie, ils se sont basés sur une méthodologie particulière, la méthode SABiO et non en suivant les recommandations émises par Noy et McGuinness [46]. Ceci ne pose pas de problème en soit car, comme il a déjà été dit, il n'existe pas de méthode toute faite pour créer une ontologie. Cependant, il est judicieux de noter que la méthode SABiO a été mise au point par de Almeida Falbo, un des auteurs du chapitre étudié. L'ontologie proposée se base sur l'ontologie UFO [34] qui a été créée afin de proposer l'équivalent d'une ontologie fondamentale [23] classique au domaine de la modélisation conceptuelle. Un des buts poursuivis par ce papier est de permettre aux développeurs devant utiliser plusieurs langages de programmation au sein d'un même projet de facilement avoir la syntaxe et la sémantique correspondantes dans les différents langages étudiés. Ces langages sont les suivants :

— Smalltalk

— Java

— Eiffel

22. Un parser est un robot/outil qui va passer en revue tous les éléments d'un texte, d'une page web, etc d'un point de vue sémantique.

23. Une ontologie fondamentale proposer les concepts de base requis à la construction d'une ontologie de domaine [34]

— C++

— Python

Les auteurs, en se basant sur [47], déclarent que les 4 grands points constituant un langage orienté objet sont l'abstraction, l'encapsulation, l'héritage et le polymorphisme. Leur ontologie va donc traiter ces 4 notions afin de les représenter d'une manière indépendante d'un langage de programmation. Pour chacun de ces principes, des exemples sont donnés dans chacun des langages étudiés. Le travail présenté a été testé et vérifié en instanciant chacun des langages étudiés ainsi qu'en s'assurant de la cohérence de l'ontologie.

L'ontologie résultante de leur travail semble pertinente et mène à des pistes de réflexions qui seront étudiées plus tard dans ce mémoire, comme le fait de s'intéresser à d'autres langages.

A présent, nous allons nous intéresser à l'héritage dans chacun des langages étudiés pour ce mémoire.

4 L'héritage dans les différents langages étudiés

L'héritage est un mécanisme permettant de représenter, en théorie, une relation de spécialisation [24] bien que dans la pratique, il peut avoir d'autres usages tels que la réutilisation de code, déviant ainsi de son but premier. Les langages orientés objet ont tendance à mélanger les notions de sous-typage et d'héritage de telle manière que si une classe hérite d'une autre, alors elle est également un sous-type de cette classe.

Habituellement, l'héritage permet aux sous-classes de redéfinir le comportement de méthodes ce qui a pour conséquence que le sens des méthodes redéfinies peut drastiquement changer par rapport au sens que la classe parente avait défini. Il n'y a généralement pas de contraintes imposées par les langages mais le développeur peut utiliser le principe de substitution de Liskov afin de maintenir une certaine cohérence dans le programme. Ce principe assure d'ailleurs qu'un sous-type peut être utilisé là où un super-type est attendu. Pour ce faire, des préconditions et postconditions seront appliquées à chaque méthode (si cela est pertinent) assurant ainsi que les sous-classes se comporteront de manière logique avec leur(s) super-classe(s). Outre cela, le principe de Liskov ajoute des conditions sur la signature des méthodes redéfinies comme l'utilisation de types covariants, lorsqu'il s'agit du type de retour ou contravariants pour les types des arguments [8].

L'héritage peut véhiculer différents sens et catégoriser les divers types d'héritage qu'il est possible de rencontrer peut s'avérer complexe. En fonction des sources, on peut avoir des catégories très précises ou au contraire assez génériques. Par exemple, [40] présente 13 manières de classer les usages de l'héritage alors que [29] les regroupe sous 3 définitions : « subtype inheritance », « Parameterized type inheritance » et « selective inheritance ». Dans ce travail, nous nous focaliserons sur la notion d'héritage au sens large du terme. Selon sa définition du « subtype inheritance », un type est un sous-type d'un autre lorsque le principe de Liskov énoncé précédemment est respecté.

Le but de cette section est de présenter, de manière synthétique, le fonctionnement et les particularités de l'héritage dans chacun des langages étudiés. Elle sera divisée en trois sous-sections :

— Les langages ne supportant que l'héritage simple

— Les langages supportant l'héritage multiple

— Les langages ne supportant pas l'héritage mais ayant un mécanisme semblable

Chaque langage sera présenté en suivant les 4 points suivants : les principes de base de l'héritage, les mécanismes de redéfinition et de dissimulation [25], le moyen de limiter la propagation de l'héritage et les notions d'interface.

4.1 Héritage simple

4.1.1 Simula

Principes de base

La première version du langage Simula, nommée Simula I, est sortie en 1962. Le langage s'inspirait alors d'ALGOL 60 et a pour objectif de permettre la simulation de situations réelles. En 1967, Simula I subit de grands changements et devint Simula 67, la version la plus connue. Parmi ces

24. L'expression "is-a" est souvent utilisé pour décrire cette relation

25. Overriding et hiding en anglais

changements, on dénombre l'ajout de classes, la notion d'héritage, de méthodes virtuelles, ... Simula 67 est le premier langage de programmation à avoir introduit l'héritage et a fortement influencé les langages utilisant le paradigme orienté objet. C'est pour cette raison que ce langage sera décrit plus en détail que les autres.

L'héritage tel que proposé par Simula 67 est un héritage simple, où les sous-classes ont accès aux membres définis dans la super-classe (si la visibilité le permet) et où une super-classe peut posséder des méthodes virtuelles que les sous-classes peuvent implémenter. Ces méthodes doivent être spécifiquement marquées comme virtuelles. Elles peuvent contenir une implémentation mais cela n'est pas requis. Cependant, appeler une méthode qui ne possède pas d'implémentation résulte en une erreur. L'héritage est transitif, ce qui veut dire que les sous-classes ont également accès aux membres de leur super-classe et ce récursivement. Une même classe peut être la classe parente de plusieurs sous-classes.

De par l'utilisation de sous-classes, Simula 67 permet donc de faire du polymorphisme, celui lié à l'héritage en particulier puisque les autres types de polymorphisme ne sont pas supportés.

La manière dont Simula gère l'héritage revient à concaténer les différentes classes appartenant à un même arbre d'héritage, la classe plus enfant étant la classe la plus intérieure. Le langage utilise les notions d'*inner* et d'*outer* pour décrire les classes en fonction de leur place dans l'arbre, *inner* représentant une sous-classe et *outer* une classe parente. Simula permet aux classes parentes d'imposer où le code de la classe enfant s'exécutera, si une telle classe est présente, via le mot-clé *inner* [50].

```
begin
   class Parent;
   begin
      OutText("Before inner");
      OutImage;
      Inner;
      OutText("After inner");
      OutImage
   end--of--Parent;

   Parent class Child;
   begin
      OutText("In the inner class");
      OutImage
   end--of--Child;

   new Parent;
   new Child
end**of**program
```

FIGURE 3 – Exemple du mot-clé *inner* [50]

Malgré le fait que Simula 67 propose déjà des concepts innovateurs pour l'époque, le langage manque encore de certaines notions pratiques qui seront introduites dans d'autres langages, telles que les classes abstraites ou la possibilité d'utiliser de l'héritage multiple.

Redéfinition et dissimulation

Les sous-classes peuvent redéfinir des méthodes marquées comme virtuelles, qu'elles possèdent une

implémentation ou non. Il est également possible de redéfinir des *labels* qui permettent de voyager à travers le code (similaire au "go to" présent dans certains langages). Pour redéfinir une méthode, il suffit de la redéclarer avec les mêmes paramètres et de l'implémenter. Il est possible de redéfinir une méthode qui est elle-même redéfinie par la classe parente. Lors de l'appel à une méthode virtuelle, l'implémentation la plus concrète sera choisie.

Il est possible de dissimuler des éléments appartenant à la classe parente dans la classe enfant, telles que des méthodes ou des variables. Cela aura comme conséquence que chaque bloc constitué par une des classes possède sa propre définition desdites variables ou méthodes. De ce fait, les valeurs des variables ne seront plus forcément les mêmes en fonction de si la méthode est appelée dans la classe parente ou dans la classe enfant [50].

Mécanisme limitant la propagation
Il semblerait que Simula ne possède pas de tel mécanisme et ne peut donc marquer explicitement une classe comme étant inhéritable.

Interfaces
La notion d'interface n'existe pas dans Simula, ce qui fait que le langage ne supporte pas le polymorphisme lié aux interfaces.

4.1.2 Java

Principes de base
L'héritage en Java est un héritage simple et transitif. Il est possible d'accéder aux membres de la super-classe à condition que la visibilité soit suffisante et une super-classe peut avoir plusieurs sous-classes. Les constructeurs ne sont pas hérités mais peuvent être appelés depuis la classe enfant. L'héritage est omniprésent dans le langage Java puisque toutes les classes héritent implicitement de la classe "Object", à condition qu'elles n'héritent pas spécifiquement d'une autre classe [7].

Redéfinition et dissimulation
La classe enfant peut redéfinir les méthodes [26] définies dans la classe parente sans que ces dernières ne soient spécialement marquées comme étant virtuelles ; chaque méthode est implicitement virtuelle, il faut user d'un mot-clé spécifique pour empêcher la redéfinition. Java possède la notion de classes abstraites qui correspond aux classes où au moins une méthode est marquée comme étant abstraite (sans implémentation). Ce mécanisme force les sous-classes à implémenter la ou les méthodes abstraites et rend la classe parente non instanciable. Si une méthode définie dans la classe enfant possède la même signature qu'une méthode de la classe parente, alors elle redéfinit la méthode de la classe parente. Cela n'est vrai que si la méthode n'est pas "static", si cela était le cas, alors la méthode de la classe enfant cacherait celle de la classe parente. La différence est principalement due au fait que les méthodes "static" sont générées au moment de la compilation (et non de l'exécution comme les méthodes d'instance), ce qui a comme conséquence que la méthode parente est toujours accessible à condition de l'appeler explicitement à l'aide d'un cast effectué sur la variable du type de la classe enfant. C'est également le cas concernant les variables qui peuvent être dissimulées mais pas redéfinies [7].

26. Avec une visibilité suffisante

Mécanisme limitant la propagation
Il est possible d'empêcher la redéfinition d'une méthode à l'aide d'un certain mot-clé du langage, "final". Il permet également au développeur d'interdire l'usage de certaines classes en tant que classes parentes. Ce mécanisme visant à limiter l'héritage, étant peut-être trop restrictif, a été affiné dans la version 15 du langage puisque trois nouveaux mots-clés ont été introduits [27]. Ces mots-clés permettent de n'autoriser que quelques classes définies à hériter de la super-classe. Il faut toutefois noter que cela n'a d'incidence que sur les classes définies au niveau de la super-classe, en d'autres termes il faut également empêcher l'héritage au niveau des sous-classes (de manière totale ou partielle) ou le laisser libre de toute contrainte [7].

Interfaces
Une interface dans le langage Java est généralement décrite comme étant un contrat que la classe s'engage à satisfaire. Là où Java ne supporte que l'héritage simple, il est tout à fait possible d'implémenter plusieurs interfaces au sein d'une même classe. Une interface peut elle-même étendre une ou plusieurs interfaces, cependant il est impossible de stopper la propagation de "l'héritage" comme cela se fait pour les classes. Les interfaces exposent généralement des méthodes (sans corps) qui devront être implémentées par la classe. Depuis la version 8 du langage, une interface peut exposer une méthode dite "static" et déjà implémentée. Il n'est alors pas possible de la redéfinir bien qu'on puisse la cacher. Outre la méthode "static", il est également possible de définir des méthodes implémentées par défaut dans les interfaces qui, elles, peuvent être redéfinies [28] [7].

4.1.3 CSharp

Principes de base
Le langage C# supporte l'héritage simple et possède les caractéristiques classiques, à savoir la transitivité de l'héritage, l'accès aux membres de la classe parente, les méthodes virtuelles pouvant être redéfinies et les super-classes pouvant avoir plusieurs sous-classes. Les constructeurs et destructeurs ne sont pas hérités, ils doivent être déclarés explicitement. Il est néanmoins possible de faire référence à ceux-ci depuis la classe enfant. Toute classe en C# hérite implicitement de la classe "Object" si elle n'hérite d'aucune autre classe [2]. Le mécanisme de l'héritage en C# est similaire à celui de Java sur plusieurs points, ce qui n'est pas surprenant puisque le but originel de C# était de proposer un "langage orienté objet polyvalent, simple et moderne"[5]. Cette description correspondant fortement à Java [29].

Redéfinition et dissimulation
Il est possible de redéfinir plusieurs éléments en C#, tels que des méthodes, des propriétés ou encore des évènements. Pour que cela soit possible, il est nécessaire qu'ils soient marqués virtuels par la classe parente et marqués comme étant redéfinis dans la classe enfant, ainsi que d'avoir une visibilité suffisante. Si un élément dans la classe enfant possède la même signature qu'un élément de la classe parente sans être marqué comme étant une redéfinition, alors il cachera simplement l'élément de la classe parente. Les deux éléments cohabitent donc et celui de la classe parente reste accessible au moyen d'un cast approprié.

27. "sealed", "non-sealed" et "permits"
28. Si deux interfaces implémentées par une classe contiennent la même méthode avec toutes deux une implémentation par défaut, il devient nécessaire pour la classe de la ré-implémenter
29. La première version de C# est sortie en 2002.

C# propose également la notion de classe abstraite qui permet d'empêcher l'instanciation de cette classe ainsi que de forcer les classes qui en héritent à implémenter les éléments marqués comme étant abstraits [2].

Mécanisme limitant la propagation
De par le choix qui a été fait de ne pas tout rendre redéfinissable par défaut, le langage C# peut simplement offrir la possibilité de bloquer l'héritage pour une classe dans son entièreté. Il est en revanche impossible d'autoriser certaines classes à hériter d'une super-classe et d'interdire cela aux autres classes [2].

Interfaces
Les interfaces permettent d'exposer divers éléments que les classes devront implémenter. Une classe peut implémenter plusieurs interfaces à la fois et les interfaces peuvent étendre une ou plusieurs autres interfaces. Contrairement aux classes, les interfaces ne possèdent pas de mécanisme permettant d'empêcher "l'héritage". C# supporte ce qui est appelé "Explicit interface implementation". Ce concept permet en fait d'implémenter deux (ou plusieurs) éléments possédant la même signature et se trouvant dans des interfaces différentes et non liées. Ces méthodes ne seront alors accessibles que si le type de la variable correspond à une des interfaces ou via un cast. Depuis sa 8ème version, C# supporte les implémentations par défaut dans les interfaces ainsi que la définition d'éléments static et la version 9 ajoute la possibilité d'avoir un type de retour covariant pour les méthodes redéfinies. La version 11 offre quant à elle la possibilité d'exposer des méthodes "static" virtuelles (ou abstraites) qui sont donc redéfinissables dans les classes [30] [2].

4.1.4 FSharp

Principes de base
Le langage F# fait partie de la famille des langages .NET, tout comme C#. F# supporte donc un héritage simple avec accès aux membres de la classe parente et possibilité de redéfinir des méthodes dans la classe enfant. Afin que ces dernières puissent être redéfinies, il faut qu'elles soient spécifiquement marquées. Une classe peut être la classe parente de plusieurs sous-classes. Si aucune classe de base n'est définie, alors les classes héritent par défaut de la classe "Object". F# supporte l'utilisation d'"object expression". Ce concept permet de simuler l'héritage sans devoir déclarer un type. En partant d'un type déclaré, il est possible de redéfinir ou d'implémenter un ou plusieurs membres du type déclaré. Cela permet de ne pas générer des types dont l'utilisation est particulièrement spécifique.

Par exemple, si on reprend le code provenant de la documentation de Microsoft [10], on peut voir que l'utilisation d'un "object expression" permet de redéfinir la méthode "ToString()" pour une instance en particulier.

```
let obj1 = { new System.Object() with member x.ToString() = "F#" }
```

30. Ceci est en preview lors de la rédaction de ce travail et son usage serait limité aux interfaces liées aux opérateurs mathématiques

Redéfinition et dissimulation
La déclaration d'une méthode virtuelle se fait de la même manière que la déclaration d'une méthode abstraite, à la différence près que la méthode abstraite ne possède pas d'implémentation. Si une méthode est abstraite, toute la classe le devient automatiquement. Les classes enfants pourront alors redéfinir ces méthodes à condition de le spécifier explicitement. Si cela n'est pas fait et que la signature est identique, alors la méthode parente sera cachée et accessible via un cast (ou depuis la classe enfant directement).

Mécanisme limitant la propagation
Il est possible, au moyen d'un attribut, de marquer une classe comme ne pouvant pas être héritée plus en avant (ou implémenté dans le cas de classes abstraites). Il semblerait que cette manière de faire n'était pas prévue dans le langage d'origine et serait un ajout fait par la communauté [31].

Interfaces
Les interfaces F# peuvent hériter de plusieurs interfaces et déclarent des méthodes et des propriétés qui devront être implémentées par les classes qui implémentent ces interfaces. F# a pour particularité que les éléments définis dans les interfaces ne sont pas accessibles depuis l'extérieur sans passer par un casting [6].

4.1.5 Scala

Principes de base
L'héritage dans Scala est un héritage simple, transitif et où plusieurs sous-classes peuvent hériter d'une même super-classe. Il est possible d'accéder aux membres de la classe parente depuis la classe enfant. Toutes les classes sont des potentielles super-classes mais il est possible d'indiquer que certaines classes ont été conçues pour être étendues (sans les rendre abstraites). Scala est un langage qui partage de nombreux concepts avec Java [32] mais qui possède également ses subtilités, notamment par le fait que le langage supporte plusieurs paradigmes : l'orienté objet et la programmation fonctionnelle.

Dans Scala, tous les types (y compris les types primitifs) héritent d'une classe commune, "Any". Pour les classes représentant des types de références, il s'agit alors d'"AnyRef" [14].

Redéfinition et dissimulation
Toutes les méthodes sont par défaut redéfinissables par les classes enfants, à condition de respecter les modificateurs de visibilités habituels. Il faut explicitement mentionner qu'une méthode est une redéfinition, si une méthode possède la même signature mais n'est pas marquée comme étant une redéfinition, alors cette méthode est invalide. En effet, Scala n'autorise pas une classe enfant à cacher une méthode de la classe parente. Il est possible de redéfinir des variables dans Scala à la condition qu'elles ne soient pas mutables.

Scala supporte les classes abstraites ; il est impossible d'instancier une classe abstraite, pour pouvoir l'utiliser, il faut donc s'en servir comme classe parente [14].

31. https ://fsharp.github.io/fsharp-core-docs/reference/fsharp-core-sealedattribute.html
32. Il est par exemple possible d'appeler des routines écrites en Java dans du code Scala et inversement

Mécanisme limitant la propagation
Il est possible de marquer certaines classes comme n'étant pas dérivables à l'aide d'un mot-clé. Ce faisant, il devient impossible d'étendre la super-classe. Cette même interdiction peut être appliquée à des méthodes en particulier. Scala permet d'apporter un peu plus de finesse en marquant certaines classes ou traits comme étant "sealed". Cela aura comme conséquence que ces classes/traits ne pourront être étendus que si la classe enfant se trouve dans le même fichier que l'élément étendu. Cette restriction ne s'applique qu'aux descendants directs, si les sous-classes ne bloquent pas elles-mêmes l'héritage explicitement, alors il est possible d'en hériter comme d'une classe normale [15].

Interfaces
Il n'existe pas de notion d'interfaces en tant que telles, à la place Scala utilise des traits. Ces traits se comportent comme des interfaces, en déclarant des méthodes que les classes devront implémenter. Ils peuvent spécifier une implémentation par défaut et une classe peut étendre plusieurs traits à la fois. Depuis la version 3 du langage, les traits supportent les constructeurs avec argument, ce qui n'était pas le cas dans la version précédente. Pour pallier à cela, dans la version 2 de Scala, les classes abstraites étaient parfois utilisées alors qu'un trait aurait mieux convenu. Cela induisait alors une sémantique qui n'était pas forcément souhaitée. Les traits ont pour vocation de faciliter la réutilisation de bouts de code tandis que l'action d'hériter d'une classe indique clairement une volonté de suivre une relation logique et de créer des instances d'un certain type [14].

4.2 Héritage Multiple

4.2.1 C++

Principes de base
C++ est un langage multi-paradigmes qui supporte l'orienté objet. L'héritage présent dans le langage C++ peut être vu de deux manières différentes, il s'agit soit d'un héritage simple soit d'un héritage multiple. Dans le cas d'un héritage simple, le fonctionnement est classique, avec son héritage transitif, l'accès aux membres de la classe parente et le fait qu'une super-classe peut avoir plusieurs sous-classes. La relation exprimée par un héritage simple est celle du "is-a" [33] [16].

Lorsqu'il s'agit d'héritage multiple, la classe enfant déclare toutes les classes dont elle hérite et ce dans un certain ordre. Cet ordre peut avoir de l'importance car c'est lui qui définit l'ordre d'appel des constructeurs et des destructeurs des classes parentes. Il n'est pas possible d'hériter deux fois directement d'une même classe mais il est tout à fait autorisé d'hériter indirectement plusieurs fois de la même classe. Un exemple typique où cela se produit est le problème du diamant.

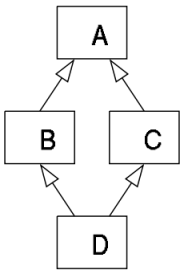

FIGURE 4 – Problème du diamant

33. ou "kind of" comme stipulé par la documentation du langage

Cette situation induit donc que la classe parente commune serait présente de multiples fois dans le corps de la classe enfant (lors de l'instanciation de la classe enfant). Il est possible d'empêcher ce comportement grâce à un mot-clé et de n'avoir qu'un seul objet commun à toutes les classes. L'héritage multiple apporte d'autres problèmes tels que les soucis d'ambiguïté, notamment concernant les méthodes se trouvant dans les classes parentes avec une signature identique. Il faudra alors expliciter quelle implémentation est à appeler. Le même problème peut être rencontré lors de casting [34], il sera alors nécessaire de spécifier le chemin complet que l'appel devra suivre. Un autre concept présent en C++ pour gérer un type d'ambiguïté est la "Dominance". Si deux éléments possèdent le même nom (une variable et une méthode par exemple) dans deux classes liées par l'héritage, alors c'est l'élément de la classe dérivée qui aura la dominance lors d'un appel d'une sous-classe [9].

```
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};

class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

FIGURE 5 – Exemple du concept de Dominance [9]

Une particularité de C++ au niveau de l'héritage, simple comme multiple, est qu'il est possible d'attribuer une visibilité spécifique à chacune des super-classes. Cette action va permettre de modifier la visibilité des éléments provenant de la super-classe mais seulement de manière restrictive. Il n'est pas possible d'exposer quelque chose qui n'était pas destiné à l'être dans la classe parente, il est cependant possible de cacher au reste du monde des éléments qui étaient visibles [1].

Il n'y a pas de super-classe par défaut. Une classe peut donc n'avoir aucune super-classe.

Redéfinition et dissimulation

La notion de "virtual" est quelque peu différente en C++ par rapport aux autres langages. En effet, toutes les méthodes peuvent être redéfinies mais celles qui sont marquées expressément comme étant virtuelles permettent deux choses. Premièrement, de pouvoir expliciter une redéfinition (via un mot-clé introduit dans la version 11 du langage). Ceci n'est pas obligatoire et n'ajoute rien au point de vue sémantique mais est surtout intéressant pour le maintien de code et la lisibilité. Deuxièmement, de définir quelle implémentation sera appelée. L'appel à une méthode virtuelle ira toujours chercher l'implémentation appartenant au type effectif alors qu'un appel à une méthode non virtuelle se basera sur le type de la variable pour savoir où se trouve la méthode à exécuter.

34. Notamment lorsque l'instance de la super-classe n'est pas partagée par les deux super-classes intermédiaires

Les variables ne peuvent être déclarées virtuelles. Une classe abstraite en C++ est une classe qui comporte au moins une méthode comme étant "pure virtual". Ceci implique que les classes qui hériteront devront implémenter ces méthodes ou devenir également abstraites. Il est impossible d'instancier une classe abstraite.

Il n'existe pas vraiment de mécanisme pour cacher des méthodes comme c'est le cas dans d'autres langages parce que le concept décrit ci-dessus rend cela inutile. Il est toutefois possible de cacher des variables [1].

Mécanisme limitant la propagation
Il est possible de marquer une classe entière ou une méthode particulière comme n'étant pas sujet à l'héritage. Cependant, pour qu'une méthode ne soit pas redéfinissable, elle doit être explicitement marquée virtuelle [1].

Interfaces
Les interfaces en C++ ne peuvent contenir que des méthodes "pure virtual". Une interface peut hériter de plusieurs interfaces. Il est possible de redéfinir explicitement les méthodes d'une interface, ce qui peut s'avérer très utile dans le cas où une classe implémente plusieurs interfaces avec la même signature. De cette manière, plusieurs implémentations peuvent être proposées. Les méthodes correspondant au type de la variable seront effectivement appelées, il faut donc que la variable soit du type d'une des interfaces ou castée en tant que tel [1].

4.2.2 OCaml

Principes de base
OCaml est un langage qui supporte plusieurs paradigmes, dont l'orienté objet. Contrairement aux autres langages orientés objet traditionnels où une classe est généralement équivalente à un type, en OCaml deux classes peuvent produire des objets du même type. Il est également possible de créer des objets n'appartenant à aucune classe, on parlera alors d'objets immédiats. Cette manière de faire a ses avantages et ses contraintes. Une de ces contraintes est qu'il est impossible d'utiliser l'héritage avec de tels objets.

En ce qui concerne l'héritage en lui-même, c'est un héritage multiple transitif et où une super-classe peut avoir plusieurs sous-classes. L'ordre dans lequel les classes sont listées lors d'un héritage multiple a de l'importance. En effet, c'est la classe listée en dernière position qui aura la précédence sur le reste. De ce fait, les méthodes et variables présentes dans la dernière super-classe redéfiniront celles définies dans les autres super-classes. Il reste néanmoins possible d'appeler explicitement un membre d'une super-classe spécifique. Toutes les méthodes et variables peuvent être redéfinies dans les sous-classes.

Les classes ne sont pas des sous-classes d'une super-classe implicite. Si aucun héritage n'est renseigné, alors la classe n'hérite de rien [11].

Redéfinition et dissimulation
Pour redéfinir une variable ou une méthode, il suffit de reprendre sa signature et de lui donner une nouvelle implémentation ou valeur. Il est possible d'annoter une méthode (ou une variable) comme étant une redéfinition mais cela n'est en rien une obligation. Néanmoins, dans le cas où la

méthode ne serait pas présente dans une classe parente, une erreur indiquera que la méthode ne redéfinit rien. Il est à noter que la redéfinition de variable a été introduite dans la version 3.10 du langage. Auparavant, les variables étaient cachées [13].

Il n'existe pas de moyen de cacher une variable ou une méthode, tout est redéfini.

Les classes abstraites sont supportées en OCaml et peuvent comporter une ou plusieurs méthodes dont l'implémentation est laissée aux classes enfant. Une classe abstraite n'est pas instanciable [12].

Mécanisme limitant la propagation
Il semblerait qu'il n'y a pas de mécanisme permettant de stopper la propagation de l'héritage.

Interfaces
Les interfaces (appelées communément interface de classe) permettent d'exposer un type défini et de déclarer les méthodes qui devront être implémentées. Puisqu'une interface correspond à un type et est liée à une classe en question, une classe n'aura toujours qu'une seule interface. Il n'est d'ailleurs pas nécessaire de la déclarer soi-même car elle peut être inférée depuis sa classe mais si la déclaration est faite, alors elle peut servir à restreindre le type créé. Les interfaces n'ont donc pas vraiment le même but que celles dans les langages plus traditionnels (orienté objet) [12].

4.3 Pas d'héritage

4.3.1 Haskell

Principes de base
Le langage Haskell est un langage purement fonctionnel. Puisqu'il ne supporte pas le paradigme orienté objet, l'héritage comme on l'entend habituellement n'est pas présent. Ceci dit, il est possible de le simuler, du moins en partie. Haskell utilise des "typeclasss" qui agissent en quelque sorte de la même manière qu'une interface dans un langage orienté objet classique (C# ou Java par exemple). Un type (qui correspond à la notion de classe) peut donc implémenter une ou plusieurs "typeclass". Il existe cependant quelques différences, comme le fait qu'il ne faille pas obligatoirement implémenter toutes les méthodes. C'est notamment le cas lorsque les méthodes exposées sont liées et où le résultat d'une de ces méthodes permet de déduire le résultat de la seconde. On parlera alors de « définition complète minimale »[4].

Pour donner un exemple, en se basant sur la Figure 6, on peut se servir de la classe "Eq" qui permet de définir deux opérateurs, l'égalité (==) et la non-égalité (/=). Les deux premières lignes qui suivent la classe "Eq" indiquent la signature attendue des opérateurs. Dans notre cas, on attend deux objets en paramètre et on retournera une valeur booléenne. Les lignes suivantes montrent que le résultat d'un des opérateurs est égal à la négation de l'autre. De ce fait, implémenter un seul des opérateurs est suffisant [4].

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

FIGURE 6 – Exemple d'une typeclass [17]

Redéfinition et dissimulation
Les typeclass représentent un moyen pratique d'user du polymorphisme ainsi que d'implémenter les méthodes.

Puisqu'il n'y a pas d'héritage, il n'y a donc rien à redéfinir ou cacher.

La notion de classe abstraite n'existe pas en Haskell mais peut néanmoins être simulée de diverses manières. Cependant, cette simulation perd toute la sémantique que les classes abstraites portent. Parmi ces manières, on retrouve notamment les typeclass [35].

Mécanisme limitant la propagation
Aucun mécanisme semblable n'existe pour Haskell.

Interfaces
Comme dit précédemment, la notion d'interface correspond, dans une certaine mesure, aux typeclass.

4.4 Récapitulatif

Les différents concepts principaux des langages étudiés ci-dessus peuvent être résumés par le tableau suivant :

	Simula	Java	C#	F#	Scala	C++	OCaml	Haskell
Héritage	Simple	Simple	Simple	Simple	Simple	Multiple	Multiple	Pas supporté
Redéfinition	Limité Explicite	Limité Implicite	Limité Explicite	Limité Explicite	Limité Explicite	Limité Implicite	Totale Implicite	Pas supporté
Dissimulation	Autorisé	Autorisé	Autorisé	Autorisé	Pas supporté	Autorisé	Pas supporté	Pas supporté
Limitation Propagation	Pas supporté	Supporté	Supporté	Supporté	Supporté	Supporté	Pas supporté	Pas supporté
Classes abstraites	Pas supporté	Supporté	Supporté	Supporté	Supporté	Supporté	Supporté	Supporté [35]
Interfaces	Pas supporté	Supporté	Supporté	Supporté	Traits	Supporté	Supporté [36]	Supporté [23]
Super-classe implicite	Non	Oui	Oui	Oui	Oui	Non	Non	Non
Paradigme orienté objet	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Non

TABLE 1 – Concepts principaux des langages

35. Peut être émulé
36. Ne correspond pas aux interfaces traditionnelles de l'orienté objet

5 Méta-modèle

Avant de réaliser l'ontologie à proprement parler, un méta-modèle reprenant les concepts clés a été réalisé via l'outil DB-Main. Ce méta-modèle suit les principes des schémas entités-relations, toutefois avec quelques largesses.

Agencer les différents concepts qui font partie intégrante de l'ontologie de cette manière permet de mettre en exergue les différents liens liant les concepts et cela apporte une vision d'ensemble qu'il aurait été plus compliqué d'obtenir en utilisant seulement une ontologie.

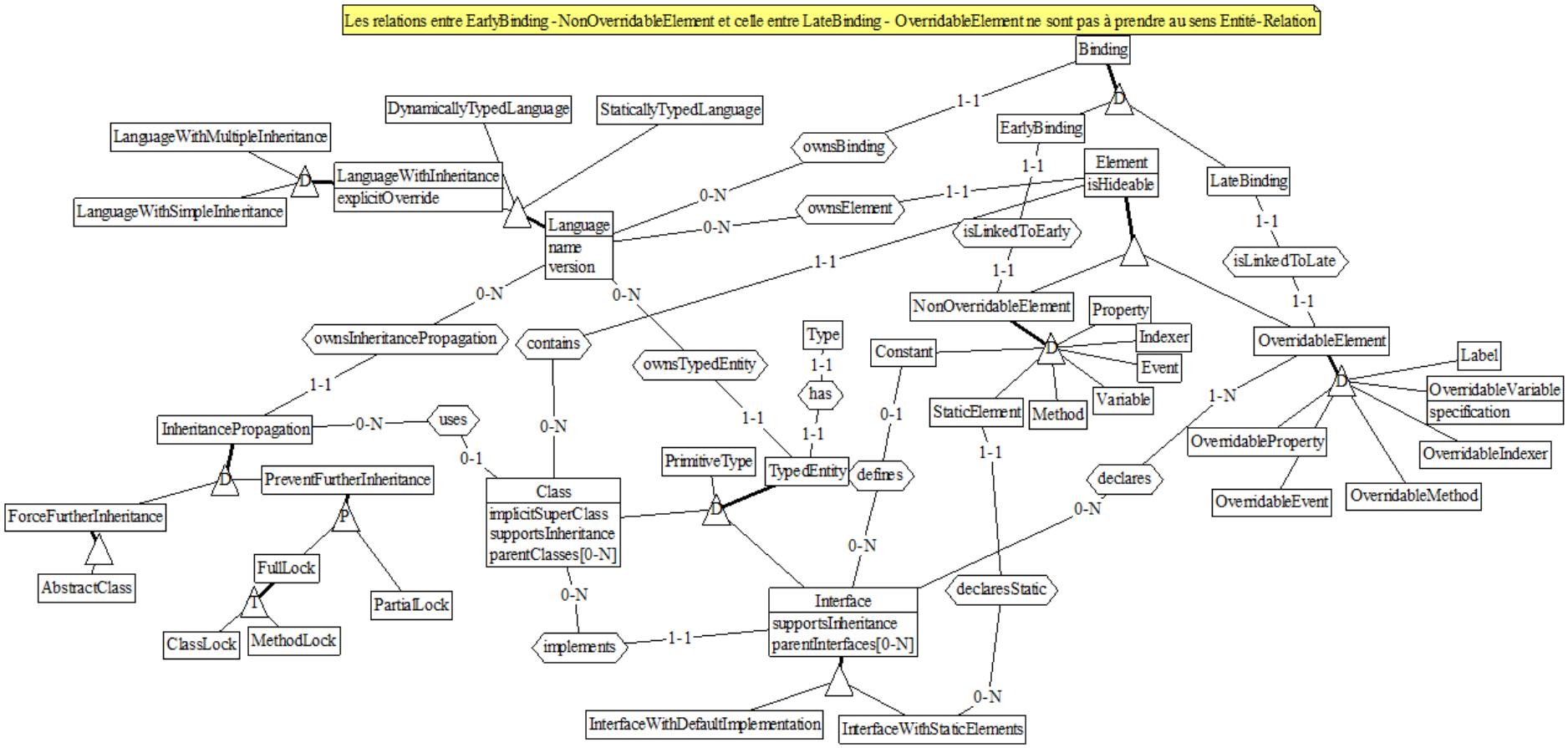

FIGURE 7 – Méta-modèle générique

5.1 Présentation du méta-modèle

Dans cette section seront définis les différents termes qui constituent le méta-modèle présenté ci-avant. Les concepts jugés les plus complexes sont repris ci-dessous. Un petit exemple de code en C# ou en Java permet d'illustrer chaque concept, si cela est pertinent.

5.1.1 Language

L'entité "Language" représente le point principal du modèle puisqu'elle représente le langage actuellement étudié. Elle possède deux attributs permettant de définir le nom du langage ainsi que sa version si cela est nécessaire.

5.1.2 Type

Les types jouent un rôle important dans les langages de programmation. Dans le paradigme orienté objet, derrière chaque classe se trouve un type. Mais les types ne sont pas liés exclusivement aux classes, une interface possède un type, ainsi que les types primitifs qui ne sont pas des objets. A noter bien évidemment que cela dépend des langages car ils ont tous leur particularité. Par exemple, Scala ou encore Smalltalk n'utilisent pas de types primitifs, il n'y a que des objets.

Les différentes entités typées sont représentées par "PrimitiveType", "Class" et "Interface". L'entité "Class" possède trois attributs ; "implicitSuperClass", "supportsInheritance" et "parentClasses". "implicitSuperClass" permet d'indiquer si toute classe hérite par défaut d'une super-classe. S'il s'avère que c'est le cas, alors la classe implicite en question n'est d'application que si aucune autre super-classe n'est renseignée. Il est possible d'indiquer si les classes supportent l'héritage via l'attribut "supportsInheritance" alors que l'attribut "parentClasses" est une collection reprenant les diverses super-classes d'une classe.

L'entité "Interface", tout comme la "Class" permet d'indiquer si l'héritage est autorisé. Elle peut être également spécialisée en deux entités "InterfaceWithDefaultImplementations" et "InterfaceWithStaticElement". La première reprend les interfaces où il est possible de donner une implémentation par défaut aux méthodes qu'elle expose tandis que la seconde permet d'exposer des éléments static.

5.1.3 StaticallyTypedLanguage et DynamicallyTypedLanguage

Il existe deux moyens d'assigner un type à une variable, le faire statiquement ou le faire dynamiquement. C'est ce que représentent les concepts StaticallyTypedLanguage et DynamicallyTypedLanguage. Un langage dynamiquement typé a pour particularité qu'il supporte l'assignation de valeurs possédant des types différents à une même variable, pratique interdite dans les langages statiquement typés (sauf s'il y a une notion de sous-typage entre ces types). Ceci permet donc une plus grande souplesse pour le programmeur, au détriment de la vérification possible au moment de la compilation disponible dans les langages statiquement typés.

De nos jours, de nombreux langages ne sont plus limités à un de ces deux types et offrent la possibilité de programmer soit en utilisant des types statiques soit des types dynamiques.

Exemple
Un exemple de langage statiquement typé supportant néanmoins le typage dynamique est le langage C#, depuis la version 4.0 qui a introduit le support des types dynamiques.

```
public class Program{
    public static void Main(string[] args){
        Animal animal = new Chien(); // Typage statique
        dynamic variableDynamic = new Chien(); // Typage dynamique
        variableDynamic = "String";
    }
}
```

5.1.4 LanguageWithInheritance

Cette entité représente les langages qui permettent l'héritage, qu'il soit simple ou multiple. Elle possède un attribut "explicitOverride" qui indique si, pour qu'un élément puisse être redéfini dans la classe enfant, il faut l'expliciter.

Exemple
Afin d'indiquer la différence entre deux langages, voici deux versions des mêmes classes en Java et en C#.

Pour qu'un élément puisse être redéfini en C#, il faut explicitement le spécifier. Sinon, il sera impossible de redéfinir l'élément.

```
public class Animal{
    public virtual void crie(){ // Redéfinition autorisée explicitement
        //Implémentation
    }
}
public class Chien : Animal{
    public override void crie() { // Permet d'indiquer qu'on redéfinit la méthode crie()
        System.out.println("wouf");
    }
}
```

En Java, il n'y a pas besoin de marquer qu'un membre peut être redéfini dans la classe parente, cependant, il faut le spécifier dans la classe enfant près de la redéfinition de l'élément, à l'aide de l'annotation @Override.

```
public class Animal{
    public void crie(){ // Redéfinition autorisée implicitement
        //Implémentation
    }
}
public class Chien extends Animal{
    @Override // Permet d'indiquer qu'on redéfinit la méthode crie()
    public void crie() {
```

```
        System.out.println("wouf");
    }
}
```

5.1.5 Element

Le concept d'Element reprend les éléments qui constituent un langage. Ils ne sont pas tous repris dans le méta-modèle, seuls ceux qui sont potentiellement impactés par l'héritage sont présents. Element comprend un attribut booléen "isHideable" qui indique si l'élément peut être dissimulé par un élément d'une sous-classe.

Element peut être divisée en deux entités distinctes, les éléments qui peuvent être redéfinis (OverridableElement) et ceux qui ne le peuvent pas (NonOverridableElement).

Dans les éléments qui ne peuvent pas être redéfinis, on a les éléments static de manière générale ainsi que les variables de classes, certaines méthodes et les constantes. Les méthodes qui ne sont pas redéfinissables peuvent l'être pour diverses raisons. Soit ce n'est simplement pas supporté par le langage soit un facteur externe invalide une possible redéfinition, comme la visibilité par exemple. Afin de pouvoir gérer le cas de F#, on a également les propriétés, les indexeurs et les évènements qui ne sont pas redéfinissables.

Dans ceux qui le peuvent, on a principalement des éléments non static tels que les méthodes, les propriétés, les évènements ainsi que les indexeurs (qui sont donc différents de ceux définis dans F#). Il existe également une classe concernant les variables qui peuvent être redéfinies et possède un attribut permettant de renseigner une possible contrainte supplémentaire. Enfin, il y a la classe Label qui est propre à Simula67.

Exemples

Les notions d'indexeur et de propriété sont surtout destinées aux langages C# et F#.

Les propriétés représentent en général les accesseurs et les mutateurs mais écrit de manière condensée. On peut bien entendu décider de ne définir que le mutateur ou que l'accesseur d'une variable. Cependant, les propriétés permettent d'exposer plus que de simples variables. Par exemple on pourrait avoir une propriété qui expose le nom complet d'une personne et concaténant le prénom et le nom de famille.

```
public class Personne{
    private int _age;
    private stirng _prenom;
    private string _nom;

    public int Age{
        get {
            return _age;
        }
        set{
            _age = value;
        }
```

```
    }

    public string NomComplet{
        get{
            return _prenom + " " + _nom;
        }
    }
}
```

Les indexeurs permettent de récupérer un élément dans une collection en fonction d'un argument passé. Leur syntaxe ressemble quelque peu à celle des propriétés. Les indexeurs ont l'avantage qu'ils encapsulent la manière dont la collection interne est gérée, empêchant ainsi toute opération non prévue. Dans l'exemple ci-dessous, on a une classe représentant une collection de personne et l'on souhaite pouvoir chercher une personne à partir de son nom.

```
public class PersonneCollection<Personne>{
    private List<Personne> _personnes;
    public Personne this[string nom]{
        get{
            for(int i = 0; i < _personnes.Count; i ++){
                if(nom == _personnes[i].Nom){
                    return _personnes[i];
                }
            }
            return null; // On n'a pas trouvé de personne avec ce nom
        }
        // On pourrait avoir un set de défini afin de remplacer une personne dans la
            collection
    }
}
```

5.1.6 InheritancePropagation

Ce concept et ceux qui en découlent permettent de décrire les moyens qui existent afin d'empêcher l'héritage de se propager ou, au contraire, forcer le programmeur à utiliser de l'héritage pour pouvoir utiliser une certaine classe parente.

Empêcher l'héritage - PreventFurtherInheritance

Dans certains cas il peut être intéressant d'empêcher qu'une classe puisse être dérivée, notamment parce que le développeur estime que le sens de la classe ne devrait pas pouvoir être modifié ou qu'avoir une classe enfant n'a pas lieu d'être. Deux mécanismes de prévention ont été observés, soit un blocage total de l'héritage, soit une ou plusieurs classes qui ont l'autorisation d'hériter de la classe parente. Cela correspond donc aux entités "FullLock" et "PartialLock" dans le méta-modèle.

A noter que pour le blocage total de l'héritage, celui-ci peut se faire à plusieurs endroits, en fonction de ce que les langages permettent. Par exemple, on pourrait bloquer l'héritage de toute une classe ou simplement bloquer la redéfinition d'une méthode en particulier. On a donc "FullLock" qui se divise

lui-même en deux sous-entité : "ClassLock" et "MethodLock". Avoir cette granularité permettant de spécifier à quel niveau on souhaite empêcher l'héritage permet d'imposer une certaine cohérence au niveau de la classe.

Forcer l'héritage - ForceFurtherInheritance
L'entité "AbstractClass" correspond aux classes abstraites telles qu'utilisées dans les langages C# et Java par exemple. Cela permet de définir une classe de base qui ne peut se satisfaire à elle-même, obligeant donc la création de classes enfants qui héritent de cette classe de base afin de pouvoir l'utiliser.

Exemple
Voici, en Java, l'illustration des concepts expliqués ci-dessus.

Pour représenter le PartialLock, nous avons donc la classe Animal qui interdit à toutes classes d'en hériter, à l'exception de la classe Chien.

```
public sealed class Animal permits Chien{
    public void crie(){
        //Implémentation
    }
}
public final class Chien extends Animal{
    @Override
    public void crie() {
        System.out.println("wouf");
    }
}
```

Le concept de ClassLock peut être illustré de la manière suivante :

```
public final class Animal{ // ClassLock
    public void crie(){
        //Implémentation
    }
}

public class Chien { // La classe chien n'a pas le droit d'hériter de la classe Animal
}
```

Et celui de MethodLock comme ceci :

```
public class Animal{
    public final void crie(){ // MethodLock
        //Implémentation
    }
}
public class Chien extends Animal{
```

```
// La class Chien ne peut redéfinir la méthode crie() car la classe Animal ne le permet
    pas.
}
```

5.1.7 Binding

La notion de Binding se décompose en deux concepts : l'Early Binding et le Late Binding.

- Early Binding : L'Early Binding, aussi connu sous le nom de Static Binding, est le fait de lier au moment de la compilation le code qui devra être exécuté pour un certain appel (à une méthode par exemple). L'Early Binding est, entre autres, utilisé pour les éléments "static", les variables, les méthodes privées, ... en sommes, pour tous les éléments qui ne peuvent pas être redéfinis par une classe enfant.
- Late Binding : Le late Binding, aussi connu sous le nom de Dynamic Binding, est le fait de lier au moment de l'exécution du programme le code qui devra être exécuté pour un certain appel. Le Late Binding est typiquement d'application lorsqu'un langage utilise le polymorphisme et concerne principalement les éléments redéfinissables.

Il est intéressant d'étudier ces deux concepts car, par exemple, lorsqu'une méthode est redéfinie, c'est le mécanisme du Late Binding qui permettra d'exécuter l'implémentation correspondant au type concret de la variable sur laquelle la méthode est appelée (si l'on est dans le paradigme orienté objet). En contrepartie, l'Early Binding va permettre d'optimiser le code par le compilateur puisque ce sera au moment de la compilation que le code à exécuter sera déterminé.

Exemple
Pour illustrer ces deux concepts, voici un bout de code réalisé en C#.

```
public class Animal{
    public virtual void Crie(){
        //Implémentation
    }
    public static void Respire(){
        //Implémentation
    }
}
public class Chien : Animal {
    public override void Crie() {
        Console.WriteLine("Wouf");
    }
}
public class Program{
    public static void Main(){
        Animal chien = new Chien();
        chien.Crie(); // Ici c'est le Late Binding qui va chercher la méthode correcte
            qu'il faudra exécuter. Il ira donc dans la classe "Chien" au moment de
            l'exécution du programme.
        Animal.Respire(); // Ici c'est l'Early Binding qui trouvera, au moment de la
            compilation, le code qui devra tre exécuté.
```

```
    }
}
```

Difficultés
Les concepts d'Early Binding et Late Binding ne doivent pas être confondus avec les notions de langage dynamiquement typé ou statiquement typé. Ces concepts peuvent sembler similaires mais diffèrent quelque peu. En effet, un langage statiquement typé peut très bien utiliser le Late Binding. Les langages tels que Java, C# ou même C++ sont statiquement typés mais utilisent énormément le Late Binding de par leur paradigme orienté objet.

5.2 Instanciation du méta-modèle

Cette section présente les différents diagrammes d'objet correspondant à chaque langage réalisé en se basant sur le méta-modèle présenté ci-avant. Ces diagrammes d'objet simplifiés [37] reprennent les concepts qu'utilise chaque langage, mettant ainsi en avant ce qu'offre le langage en question. Pour ce faire, les informations reprises au chapitre 4 présentant les divers langages et leurs subtilités seront utilisées.

Les langages de programmation évoluent et une nouvelle version peut introduire des concepts qui n'étaient pas présents dans les versions précédentes du langage. C'est pour cette raison que les langages Java et C# possèdent chacun deux diagrammes différents correspondant à une version spécifique.

5.2.1 Simula67

Simula67, étant le premier langage orienté objet, ne dispose pas de tous les concepts présents dans le méta-modèle. Le diagramme d'objet à la Figure 8 permet donc de montrer l'absence d'interface ainsi que de moyens pour forcer ou empêcher l'héritage pour une classe donnée. On peut observer que Simula67 nécessite d'expliciter une redéfinition et ne dispose pas de super-classe commune. On remarque également que les différents éléments possibles sont dissimulables. Parmi ceux-ci, on distingue deux éléments redéfinissables, à savoir un label et une méthode de classe. En ce qui concerne les éléments non redéfinissables, ils sont au nombre de 3 : les variables, les constantes et les méthodes.

37. Simplifiés car les cardinalités ne sont pas affichées et généralement seule une instance par classe est présente.

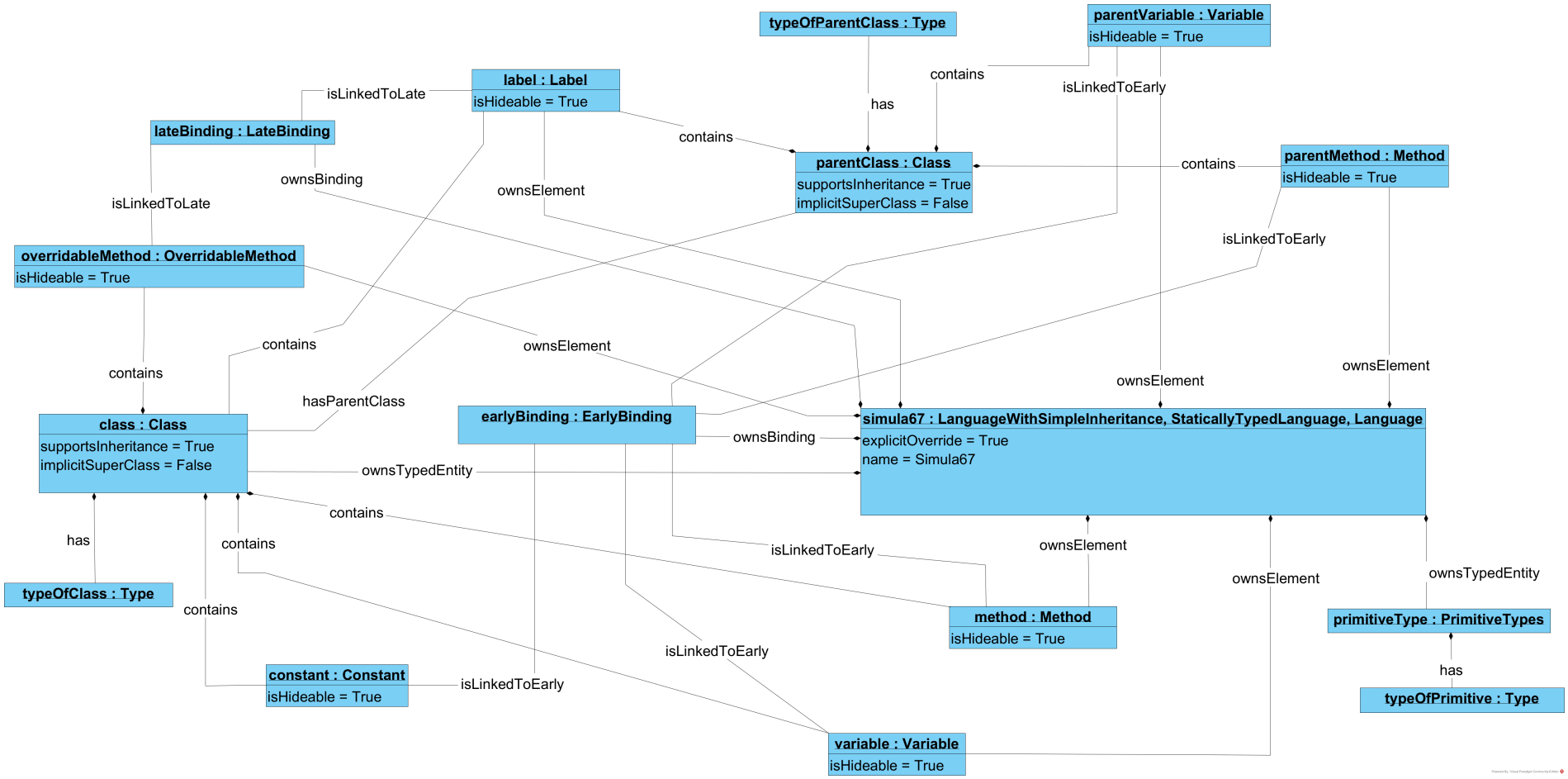

FIGURE 8 – Diagramme d'objet Simula67

5.2.2 Java 8

La première version de Java qui nous intéresse est la version 8. Cette version possède déjà la plupart des concepts étudiés. On observe, sur la Figure 9, qu'en Java, il n'est pas nécessaire de préciser qu'une méthode est une redéfinition et que toutes les classes héritent d'une super-classe commune de manière implicite. Les interfaces, tout comme les classes, supportent l'héritage. Les interfaces peuvent être classiques, contenant des éléments static et/ou contenant des implémentations par défaut pour les méthodes qu'elles déclarent. Seules les méthodes peuvent être redéfinissables, les éléments qui ne peuvent pas l'être sont composés des constantes, des méthodes, des variables et des éléments static. Chacun de ces éléments est dissimulable. Java 8 supporte les classes abstraites ainsi que diverses manières de bloquer l'héritage de manière complète, au niveau des classes ou au niveau des méthodes.

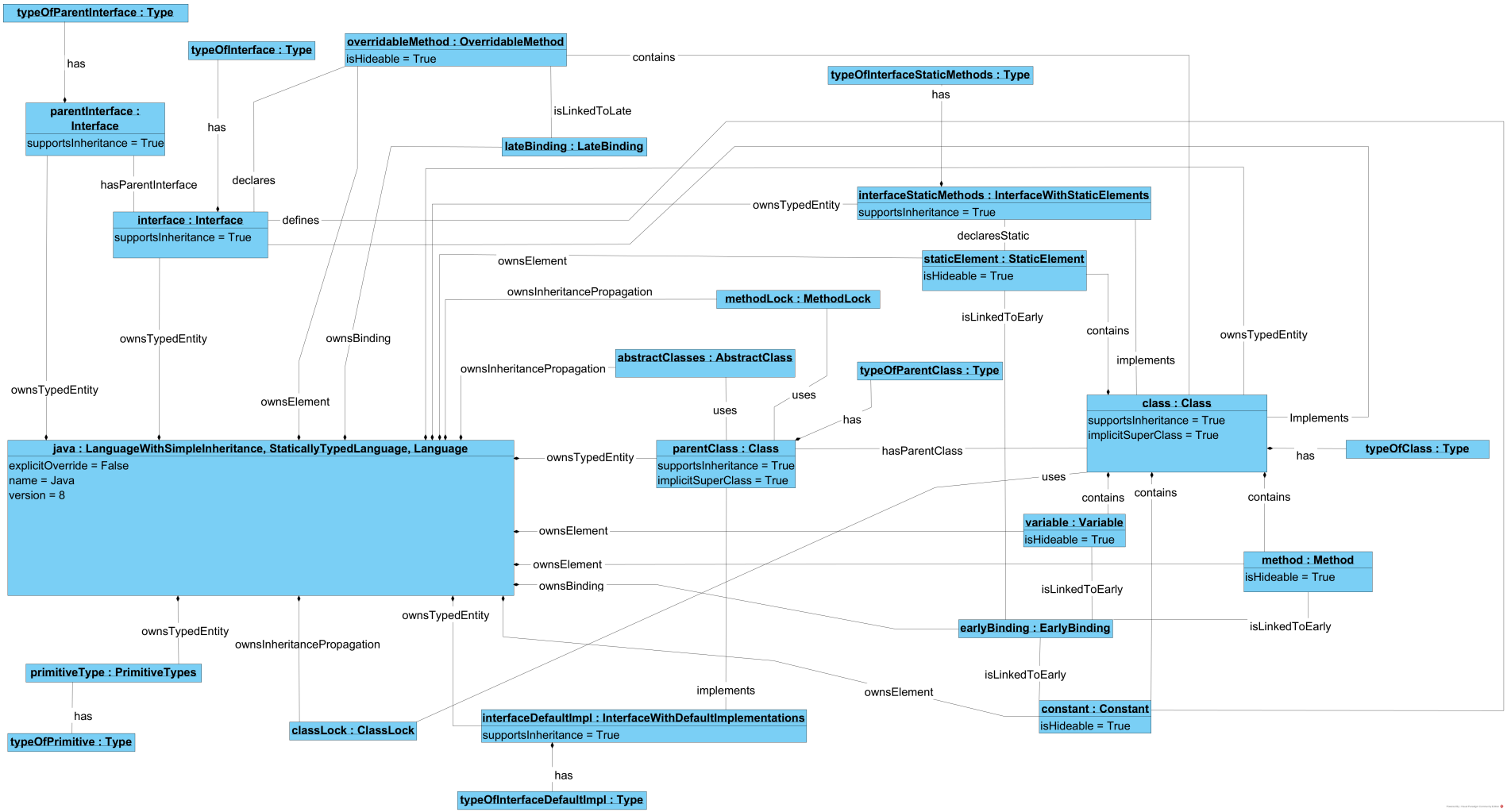

FIGURE 9 – Diagramme d'objet Java 8

5.2.3 Java 15

Java 15 garde les mêmes principes que son prédécesseur Java 8. Il ajoute néanmoins la possibilité de bloquer de manière partielle l'héritage (en autorisant donc certaines classes bien définies). Le diagramme représenté à la figure 10 ne reprend que les éléments qui changent par rapport à la version 8. C'est pour cela que plusieurs classes sont présentes, chacune étant liée à une manière de limiter ou de forcer l'héritage.

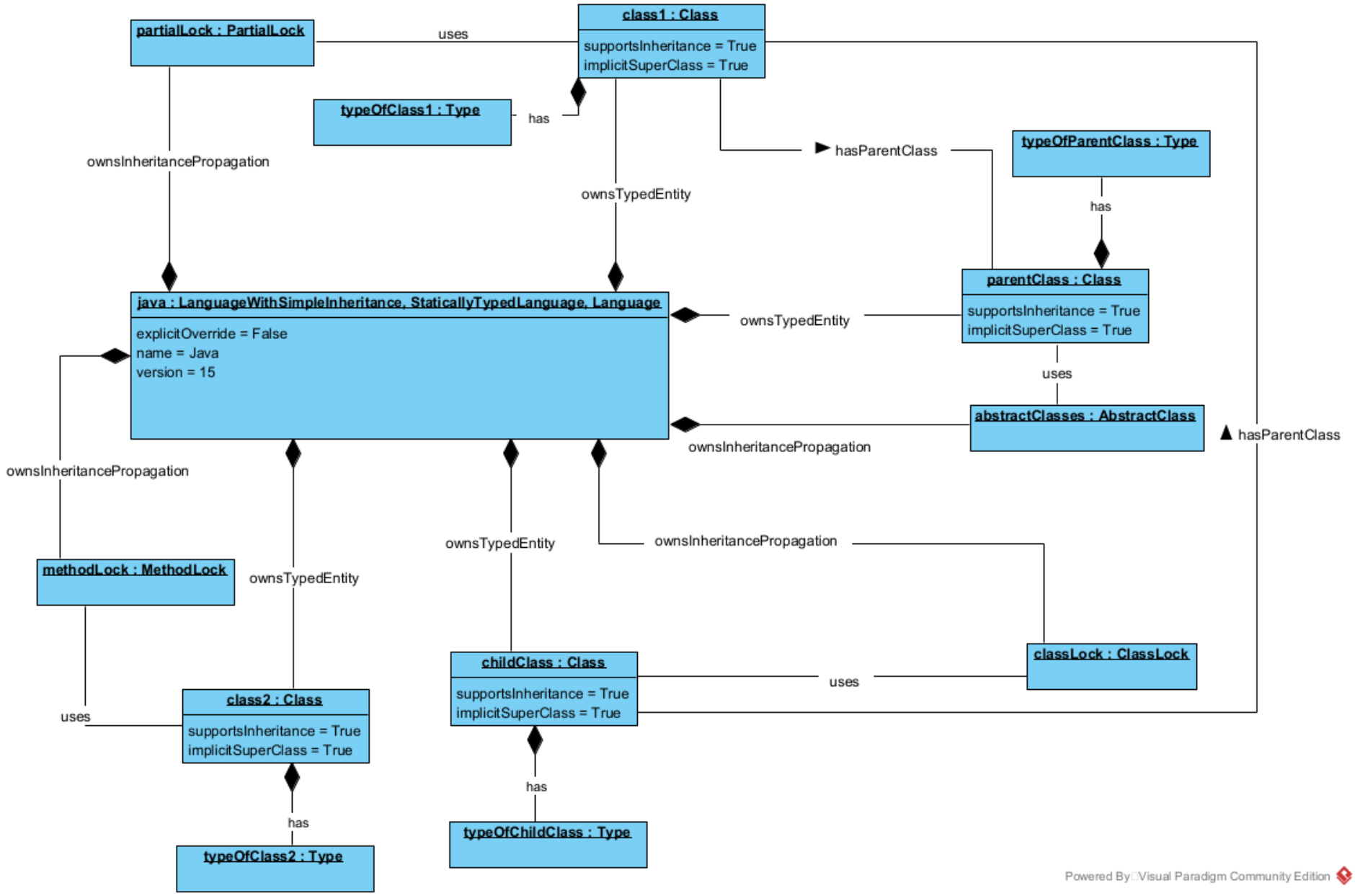

FIGURE 10 – Diagramme d'objet Java 15

5.2.4 CSharp 7

La Figure 11 présente les concepts utilisés par C# dans sa 7ème version. On peut voir que les interfaces et les classes supportent l'héritage et que les classes héritent de manière implicite d'une super-classe. C# ne permet pas de redéfinir une méthode si celle-ci n'est pas marquée comme étant sujette à la redéfinition. Les éléments qui composent le langage sont tous dissimulables. Les éléments redéfinissables sont composés des propriétés, des indexeurs, des méthodes et des évènements. Ceux qui ne le sont pas comportent des variables de classe, des méthodes, des constantes et les éléments static. On observe que les interfaces peuvent contenir des éléments redéfinissables ainsi que des constantes. Il est possible de contrôler la propagation de l'héritage via des classes abstraites ou en bloquant l'héritage d'une classe dans son entièreté. On peut noter la présence d'une super-classe qui disposes de ses propres éléments static ainsi qu'une référence vers les méthodes redéfinissables qui sont également liées à la classe enfant.

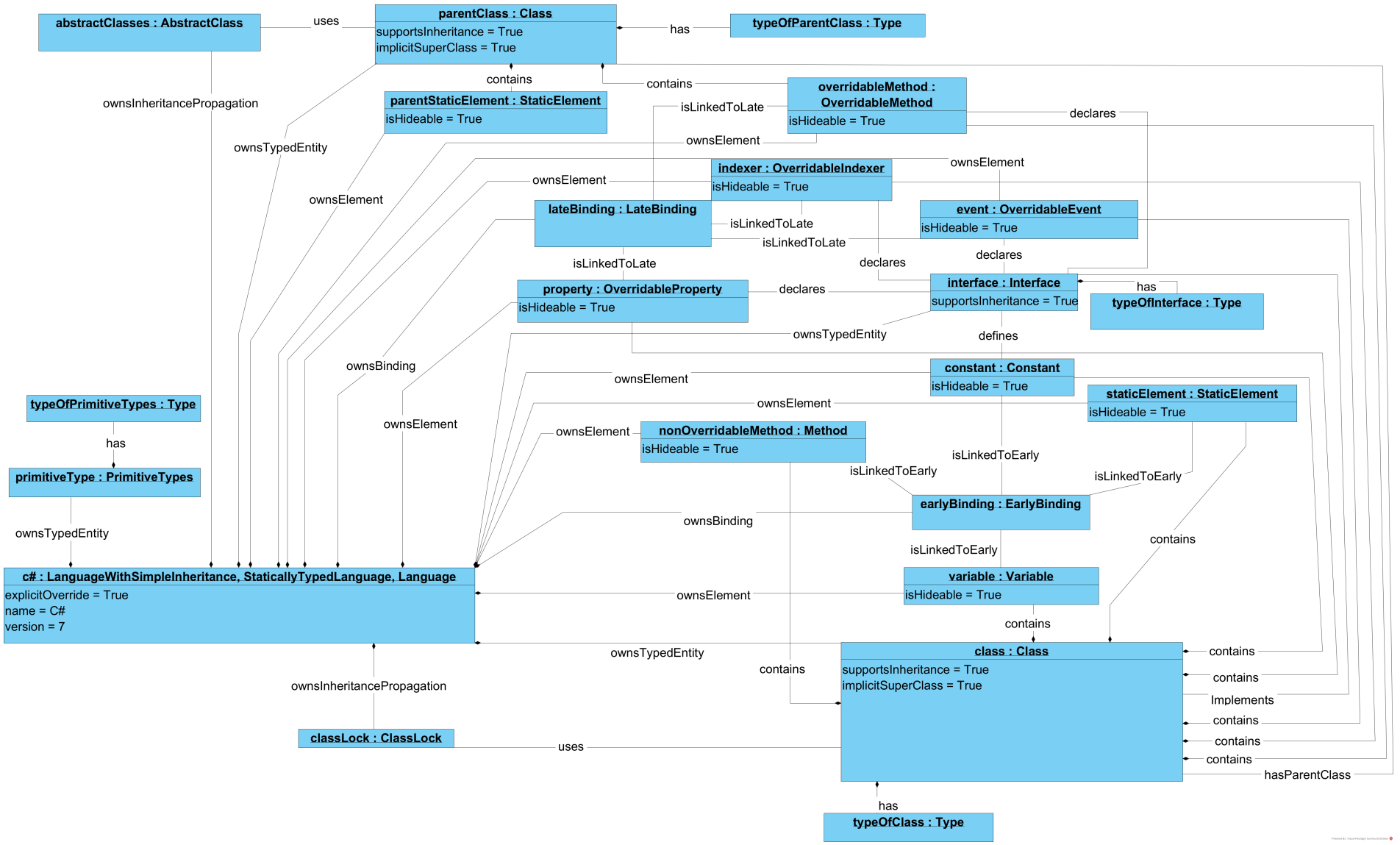

FIGURE 11 – Diagramme d'objet CSharp 7

5.2.5 CSharp 8

La Figure 12 représente C# 8, la deuxième version qui nous intéresse. Cette version reste conforme avec C# 7 et apporte comme nouveauté le support des interfaces contenant des éléments static ainsi que celles fournissant des implémentations par défaut aux méthodes déclarées.

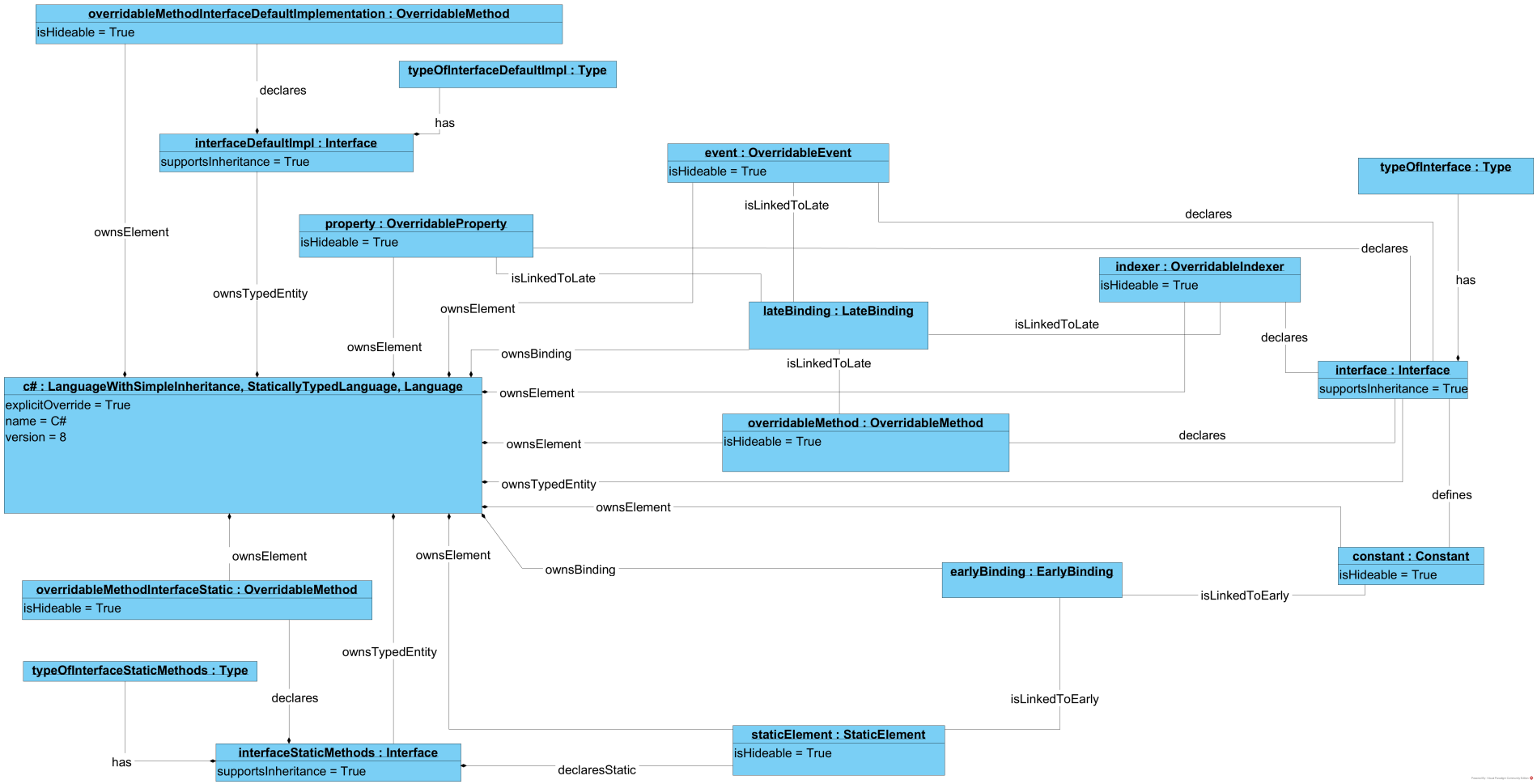

FIGURE 12 – Diagramme d'objet CSharp 8

5.2.6 FSharp

Puisque F# appartient à la même famille de langages que C# (la famille .NET), il n'est donc pas surprenant de constater que son diagramme d'objet ressemble fortement à celui de C# 7.

F# nécessite que les possibilités de redéfinition soient explicites et impose une super-classe implicite. Seules les interfaces classiques sont supportées et on observe qu'il n'est possible de bloquer l'héritage qu'au niveau des classes. Il est également possible d'user de classes abstraites pour forcer la propagation de l'héritage.

Le langage comporte deux types d'éléments redéfinissables, les méthodes et les propriétés. En ce qui concerne les éléments non redéfinissables, ils sont composés d'indexeurs, de méthodes, d'évènements, de variables, d'éléments static ainsi que de constantes. Chacun de ces éléments est dissimulable.

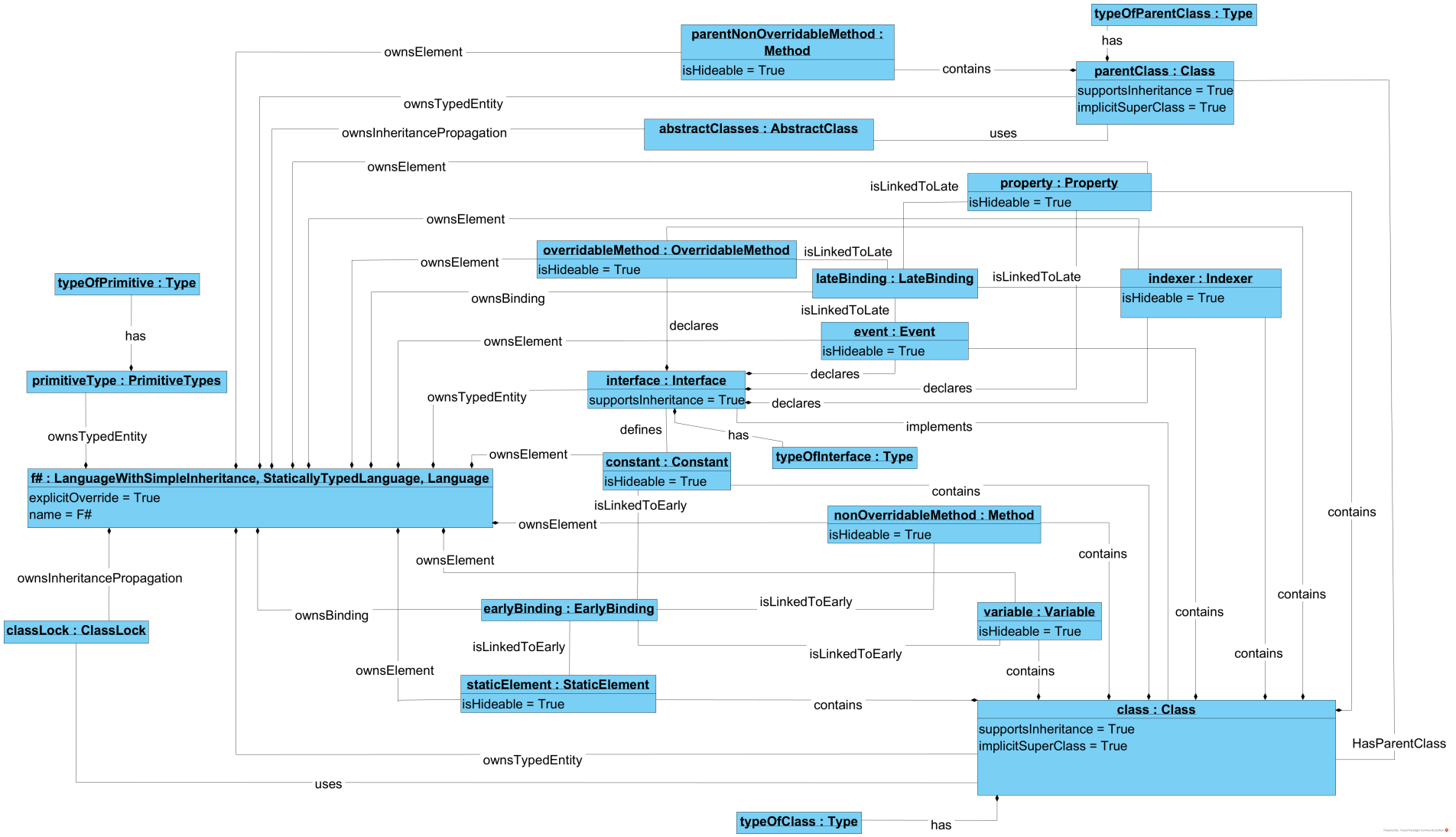

FIGURE 13 – Diagramme d'objet FSharp

5.2.7 Scala

Scala est un langage qui impose une super-classe implicite, et ce même aux types primitifs. C'est pour cette raison que ces types primitifs, qui deviennent alors de simples classes, ne sont pas repris dans la Figure 14. Outre cette particularité, la ressemblance entre Java et Scala est assez visible, notamment lorsqu'on voit que les éléments redéfinissables le sont par défaut.

Puisque Scala n'utilise pas vraiment des interfaces classiques mais plutôt des traits, il a été décidé de les modéliser comme étant des interfaces offrant des implémentations par défaut. Les classes et les traits supportent l'héritage.

Scala propose de nombreux mécanismes pour contrôler la propagation de l'héritage. Que ce soit par l'utilisation de classes abstraites ou le blocage de l'héritage. Ce blocage peut être fait pour une classe dans son entièreté ou pour une méthode particulière. Il est également possible de faire du blocage partiel comme expliqué dans le chapitre 4, dans la partie concernant Scala.

Une autre spécificité de Scala se retrouve dans les éléments proposés. Les éléments non redéfinissables sont classiques : les variables, les méthodes, les éléments static et les constantes. En revanche, en ce qui concerne les éléments redéfinissables, on observe qu'il existe deux possibilités : des méthodes ou des variables. À noter que ces dernières, afin de pouvoir être redéfinissables, ne peuvent posséder de mutateur.

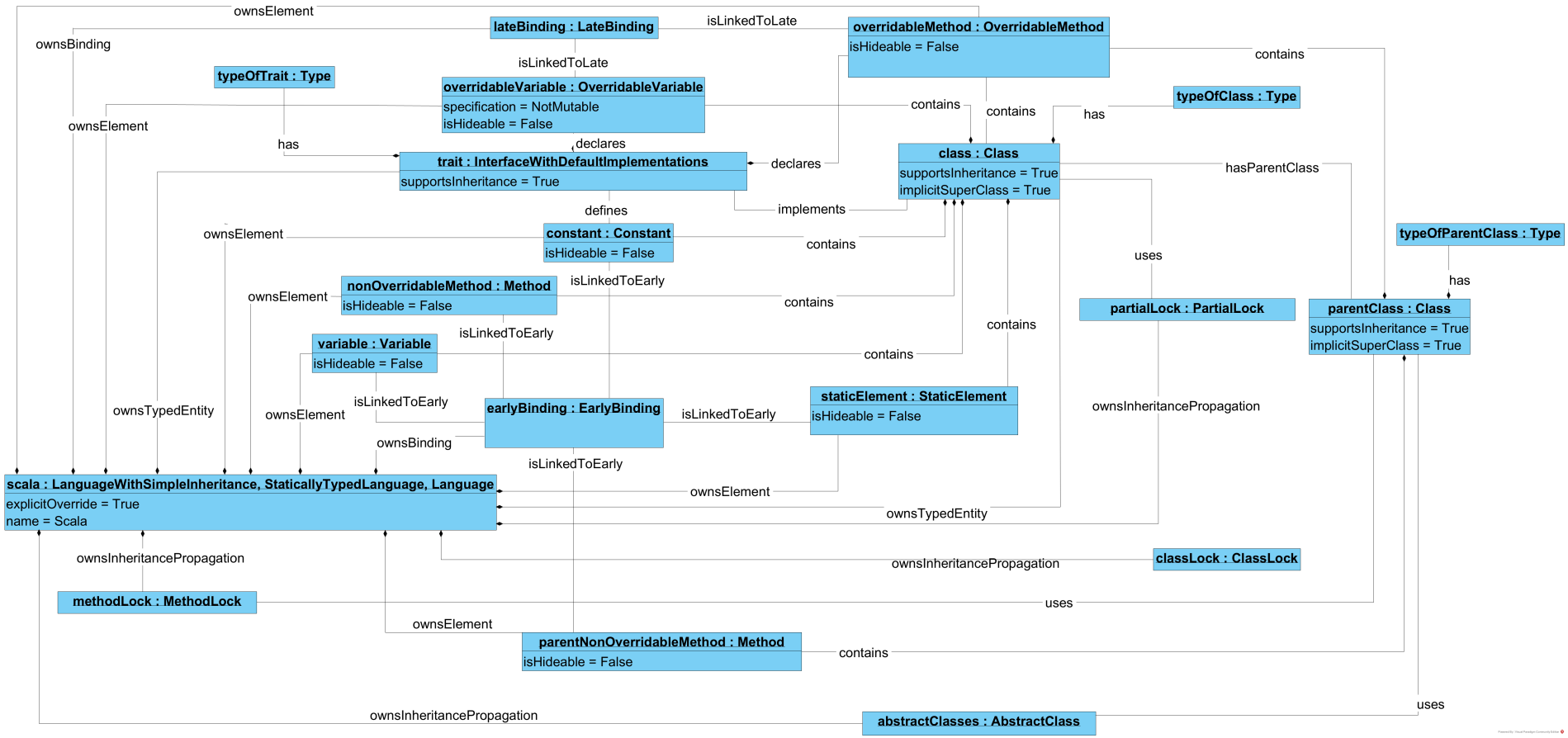

FIGURE 14 – Diagramme d'objet Scala

5.2.8 C++

C++ se distingue des précédents langages dans le fait qu'une super-classe implicite n'est pas présente. Les classes et les interfaces supportent l'héritage et il n'est pas nécessaire de préciser qu'un élément peut être redéfinissable.

C++ possède le concept de classes abstraites ainsi que blocage au niveau de la classe ou des méthodes. Seules les interfaces classiques sont supportées. Les éléments sont tous dissimulables à l'exception des méthodes auxquelles cela est interdit. Les éléments redéfinissables sont composés des méthodes et d'évènements. En ce qui concerne les éléments non redéfinissables, il s'agit des mêmes que dans les autres langages étudiés ci-avant, à savoir les variables, les méthodes, les éléments static ainsi que les constantes. Comme montré dans la Figure 15, une classe peut avoir plusieurs classes parent possédant elles-même leurs propres éléments.

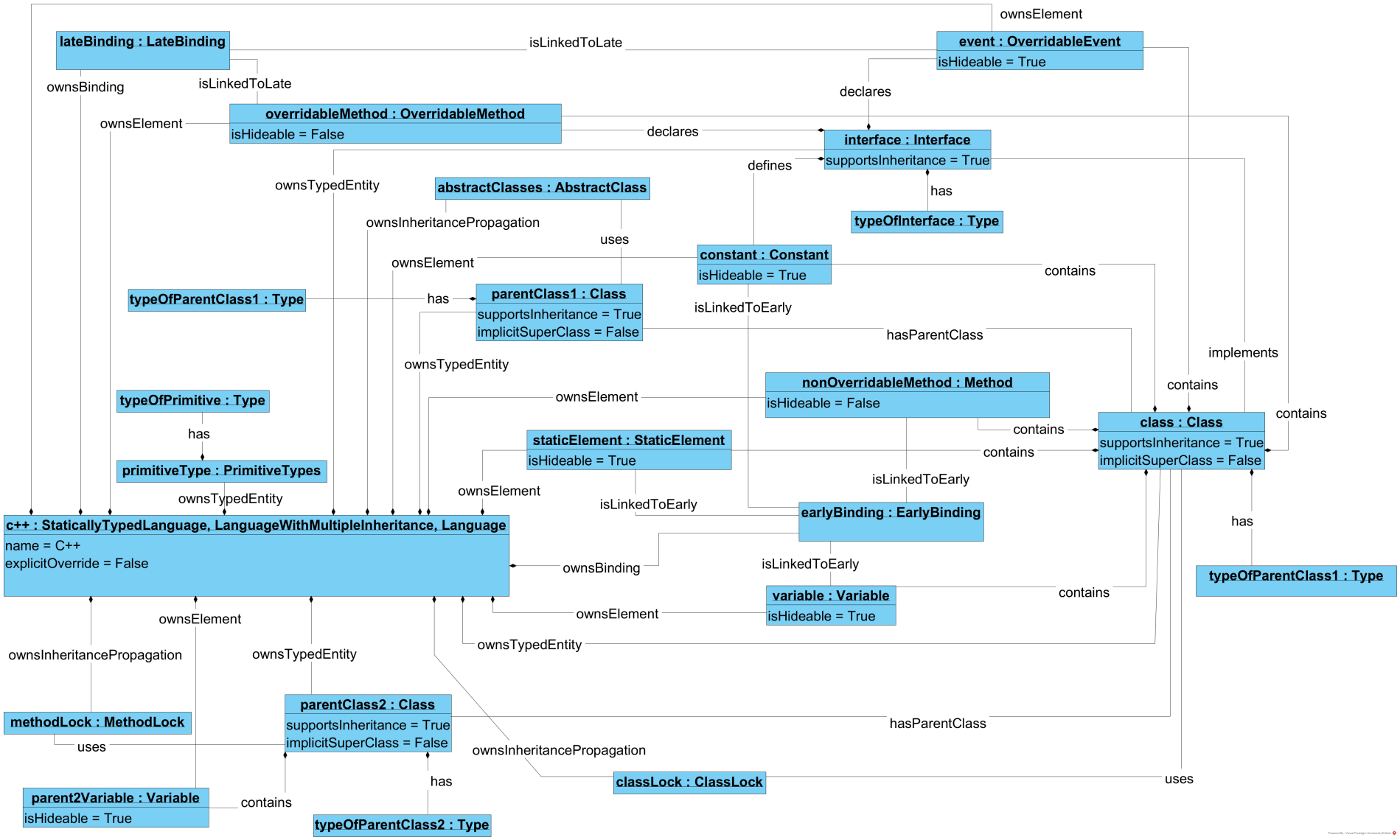

FIGURE 15 – Diagramme d'objet C++

5.2.9 OCaml

La version étudiée du langage OCaml est la 5ème car elle propose plus de concepts intéressants pour ce travail. On peut voir que les classes ne possèdent pas de super-classe implicite et qu'elles supportent l'héritage. Les interfaces, quant à elles, ne supportent pas l'héritage puisqu'une interface sert à exposer les méthodes d'un module bien spécifique. Ainsi, la Figure 16 représente une classe possédant deux super-classes qui ont chacune leur propre interface.

Il n'est pas nécessaire de marquer une méthode comme étant propre à la redéfinition et le seul moyen d'influer sur la propagation de l'héritage est via l'utilisation de classes abstraites. En OCaml, toutes les méthodes et variables sont redéfinissables. Les seuls éléments qui ne le sont pas sont les constantes.

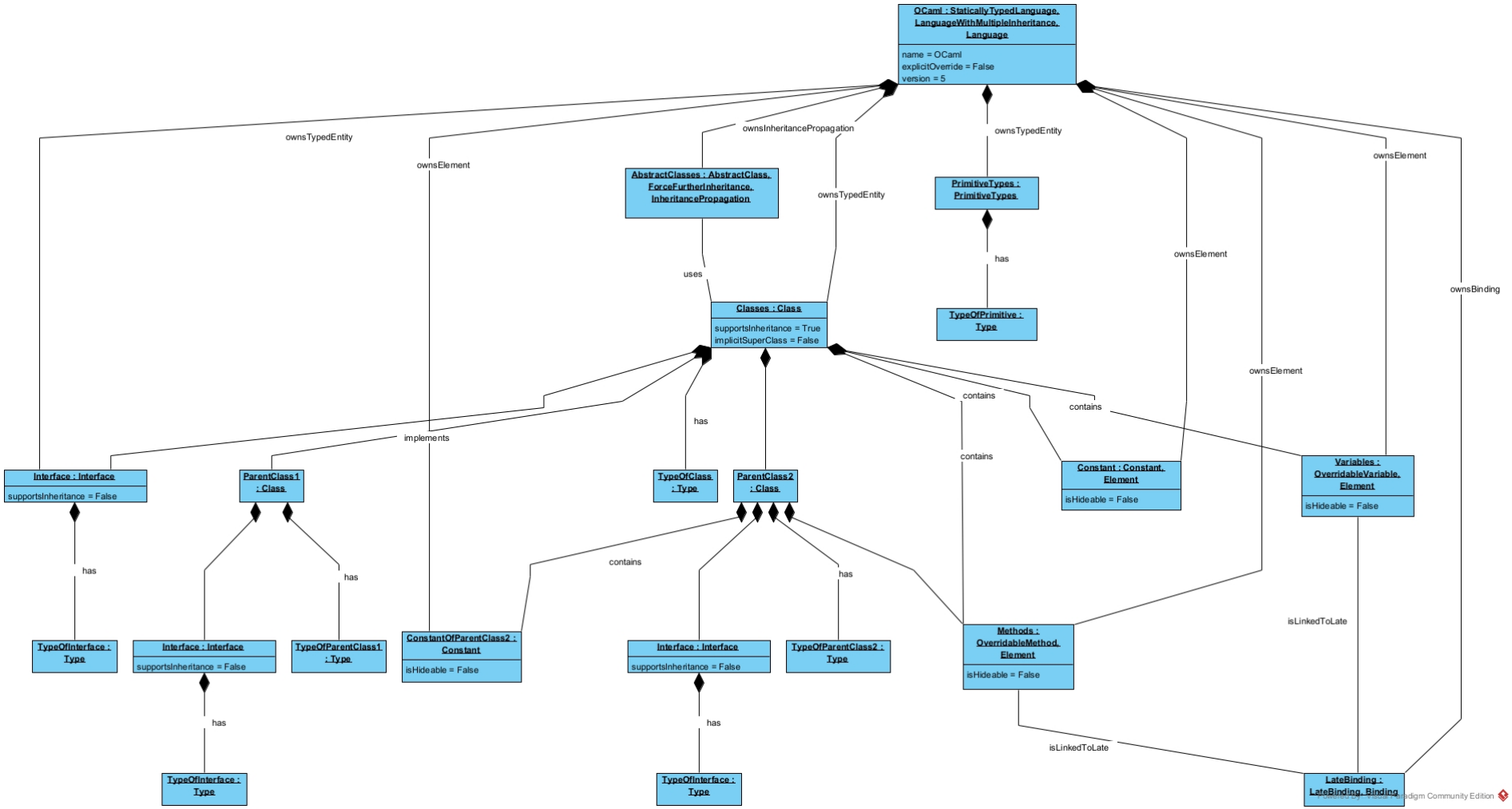

FIGURE 16 – Diagramme d'objet OCaml

5.2.10 Haskell

Haskell n'autorise l'héritage ni pour ses classes ni pour ses "interfaces". Puisque l'héritage en tant que tel n'est pas possible, il est évident qu'il n'y a pas de super-classe implicite et la question de savoir si un élément est automatiquement sujet à redéfinition ne se pose pas. Les éléments ne sont pas dissimulables et Haskell ne comporte que des éléments non redéfinissables, à savoir les constantes, les méthodes et les variables. Il existe tout de même un mécanisme ressemblant aux interfaces classiques, ce qui justifie la présence de cette instance dans le schéma.

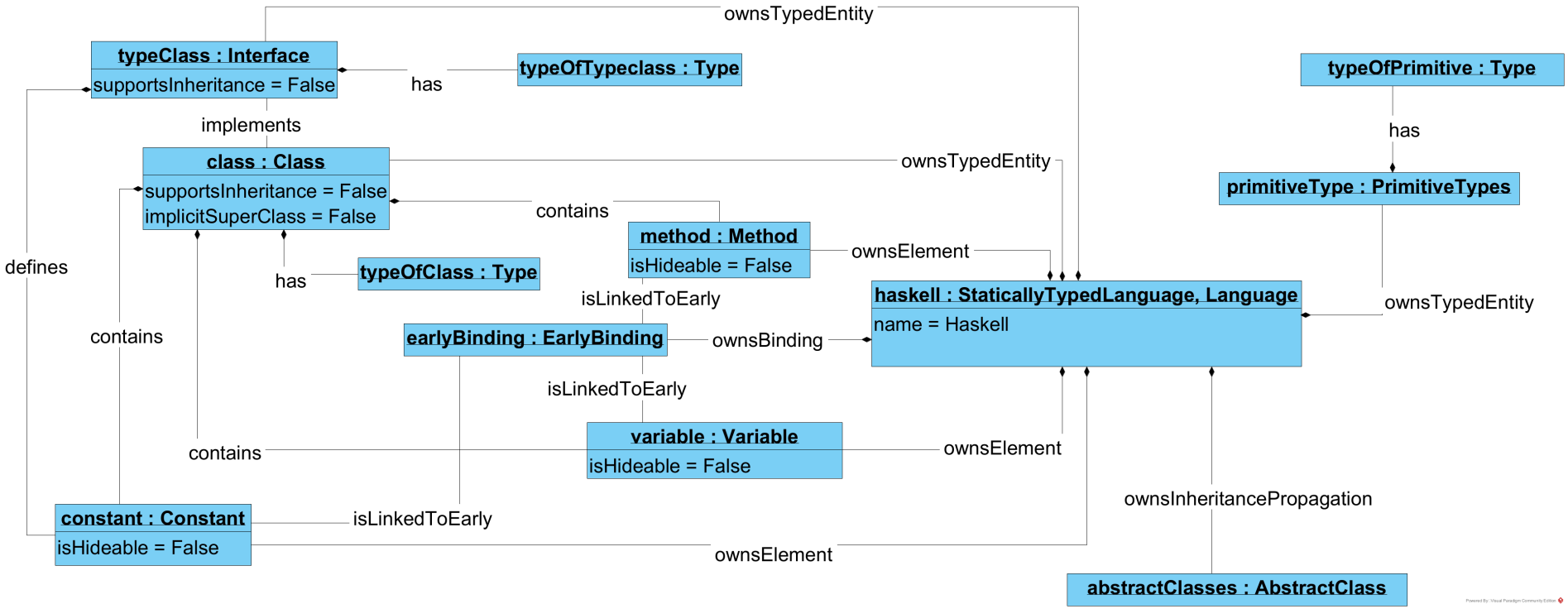

FIGURE 17 – Diagramme d'objet Haskell

6 Ontologie

À partir du méta-modèle réalisé dans le chapitre précédent, une ontologie OWL reprenant les différents concepts est créée via l'outil Protégé. Cette section a pour but d'expliquer comment l'ontologie a été développée et de présenter comment les différents langages interagissent avec l'ontologie.

La création de l'ontologie principale a suivi la méthodologie présentée par Noy et McGuinness (reprise dans le chapitre 2.4) bien que ceci ne se soit pas fait en un seul processus. En effet, la détermination du domaine et la délimitation du périmètre correspondent aux limites imposées pour ce mémoire. L'état de l'art a permis de mettre en avant les ontologies existantes et de pouvoir analyser leur structure. Les étapes concernant l'établissement des différents termes, la définition des classes, de leur hiérarchie et de leurs propriétés ont été couvertes par l'établissement du méta-modèle et son analyse. Il reste alors la définition des aspects des propriétés ainsi que la création d'instance, ce qui sera fait dans ce chapitre.

L'ontologie principale utilise la syntaxe Manchester OWL car elle est jugée plus accessible que RDF/XML et permet d'associer des descriptions à des entités. Dans un souci de clarté et de découpe, chaque langage dispose de sa propre ontologie où chacune de ces ontologies référence l'ontologie qui sert de définition commune [38].

```
Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#DynamicallyTypedLanguage>

    SubClassOf:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>

    DisjointWith:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#StaticallyTypedLanguage>
```

FIGURE 18 – Définition d'une classe via la syntaxe Manchester

Les ontologies des langages exploitent la notion d'individus, ce qui crée un parallélisme avec les diagrammes d'objet décrits précédemment. De ce fait, chaque langage est constitué d'individus basés sur l'ontologie principale.

Deux options pour représenter les langages ont été envisagées, soit une version par instance soit une version par classe. La version par instance a été retenue car elle permet d'afficher plus clairement les concepts qu'un langage utilise et comment ils s'articulent entre eux. De plus, cela renforce le lien avec les diagrammes réalisés précédemment. La version par classe, où chaque langage aurait correspondu à une classe OWL, a pour avantages de rassembler tous les langages en une seule ontologie et de pouvoir user des divers axiomes disponibles [39]. Mais puisque le sujet de ce travail est une recherche sur l'héritage d'un point de vue sémantique et non de son implémentation, les classes produites auraient été des singletons puisque n'ayant qu'une seule instance valable. Si on souhaitait étudier les différences entre chaque implémentation d'un même langage, par exemple en fonction du compilateur choisi, alors le choix entre les versions par instance et par classe mérite d'être reconsidéré.

38. Il est possible d'importer une ou plusieurs ontologies déjà existantes et de pouvoir les utiliser directement
39. Au lieu de passer par des attributs comme dans la version par instance

L'intérêt de passer par des ontologies, après la modélisation déjà effectuée, est que l'on peut se servir des moteurs d'inférence afin de valider la cohérence de l'ontologie et de mettre en avant des relations qui n'étaient pas visibles au premier coup d'oeil. Il est également plus simple d'étendre la base de connaissance que représente l'ontologie avec de nouveaux langages ou de nouvelles caractéristiques à étudier.

6.1 Ontologie principale

6.1.1 Entités

Chaque concept présent dans le méta-modèle est transcrit en une classe dans l'ontologie. La Figure 19 reprend les différentes classes qui sont utilisées. La structure est similaire à celle du méta-modèle, notamment au niveau de la hiérarchie des classes. Afin de s'assurer qu'il n'y ait pas d'inférence problématique, toutes les entités sont bien disjointes de leurs voisins directs. Il est cependant possible d'user de l'inférence pour définir de nouvelles classes sur base de relation présentes ou non. De ce fait, la classe "LanguageWithAbstractClass" est une classe qui reprend les langages où la notion de classes abstraites est présente.

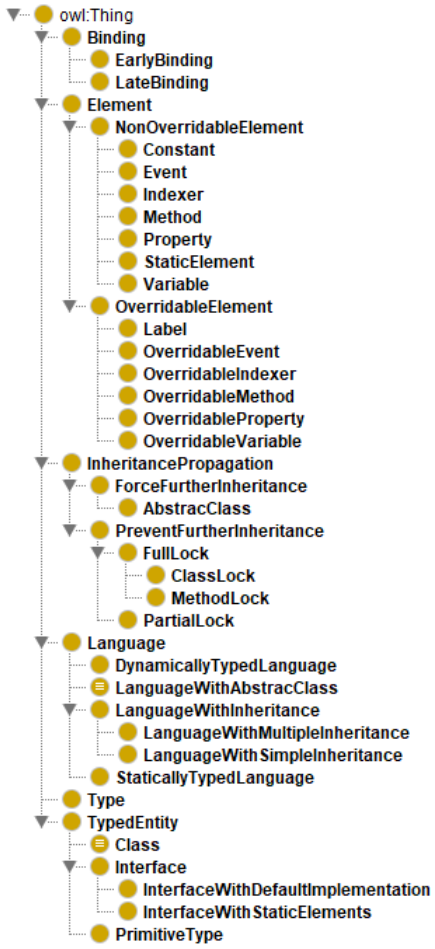

FIGURE 19 – Entités

Une classe est définie à partir de plusieurs axiomes. Ces axiomes peuvent utiliser les propriétés d'objet et de données. Les axiomes des classes enfant reprennent également les axiomes des classes parentes. Ainsi, la Figure 20 reprend la définition de la classe "Class". On observe qu'une classe dans un langage de programmation peut avoir plusieurs éléments et possède au maximum un système permettant de limiter ou de forcer l'héritage. Il est également précisé qu'une classe peut avoir une ou plusieurs super-classes et elle-même être la super-classe d'autres sous-classes. Les axiomes utilisés ici sont des conditions nécessaires et suffisantes, ce qui induit que si un individu répond à ces caractéristiques, alors il appartient à cette classe. L'utilisation de conditions nécessaires auraient simplement indiqué qu'un individu devait posséder ces caractéristiques afin d'appartenir à cette classe. L'utilisation de conditions nécessaires et suffisantes permet également d'user au maximum du moteur d'inférence puisqu'il faut qu'une classe soit définie de cette manière afin qu'il puisse déterminer quelles sont les potentielles sous-classes dans l'ontologie.

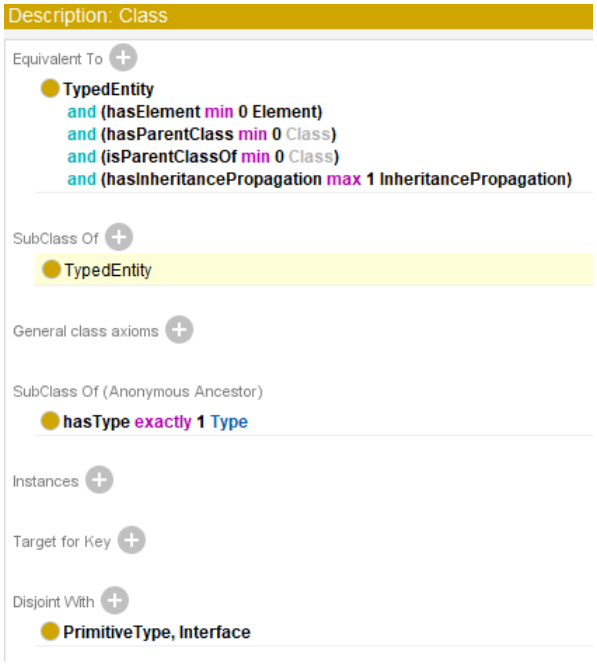

FIGURE 20 – Définition de la classe "Class"

Le code complet de l'ontologie est disponible dans l'annexe 9.1.

6.1.2 Propriétés d'objet

Les relations entre les classes du méta-modèle ont été traduites en "Object properties" (qui seront référencées désormais par le terme de "propriétés d'objet"), visibles dans la Figure 21. Pour chaque propriété d'objet correspondant à une relation déjà établie, une propriété inverse est créée. Cela est généralement considéré comme une bonne pratique et permet notamment au moteur d'inférence de sortir plus de résultats. Les propriétés d'objets suivent une convention de nommage du type "has...." et "is....Of", où les propriétés "is...Of" correspondent aux inverses des "has...". Toutes ces propriétés ont également un "Range" et un "Domain" définis. Ces valeurs correspondent à une ou plusieurs des classes présentées précédemment. Ces informations permettent de définir à quel type de classe les objets utilisant ces propriétés appartiennent mais il est important de noter que ces valeurs ne servent en aucun cas de condition. Le moteur d'inférence utilise également ces informations afin

d'être plus efficace. Certaines propriétés sont organisées d'une manière hiérarchique. Cela permet de spécifier qu'une ou plusieurs propriétés partagent un parent commun et peuvent donc être utilisées dans un axiome nécessitant la propriété "parente".

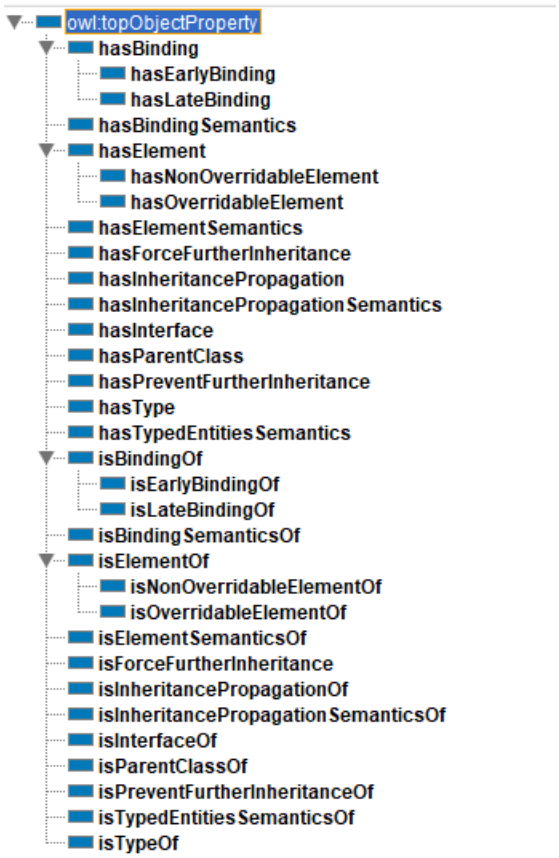

FIGURE 21 – Propriétés d'objet

6.1.3 Propriétés de données

Les attributs sont quant à eux devenus des "Data Properties" (ou propriété de données), à l'exception près du lien entre une classe et ses potentielles classes parentes qui est devenu une propriété d'objet (puisqu'elle lie un ou plusieurs individus). La Figure 22 montre les différentes propriétés créées. Il est également possible de leur attribuer un "Domain" et un "Range" pour préciser de quel type de données on parle mais cela a moins d'incidence que pour les propriétés d'objet. Les "Ranges" sont limitées aux types primitifs supportés par OWL [40] (boolean, string, int, ...).

40. Et plus particulièrement par RDF et XML

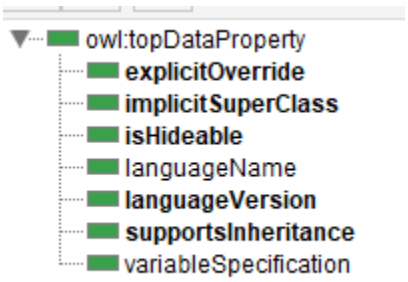

FIGURE 22 – Propriétés de données

6.2 Ontologies spécifiques

Cette sous-section a pour but de présenter les ontologies de quelques langages. La création de celles-ci se base sur les diagrammes d'objet représentant chaque langage qui sont traduits en OWL. De la même manière que le méta-modèle a été "instancié", les ontologies spécifiques se basent sur l'ontologie principale en créant plusieurs individus qui sont des instances des classes montrées précédemment. Ces individus sont liés entre eux grâce aux propriétés de données. Soit le type de ces individus est explicitement renseigné, soit les différentes propriétés d'objet liant cet individu aux autres sont suffisantes pour inférer son type. Toutes les ontologies ne sont pas présentées, seules celles qui utilisent certains concepts intéressant à présenter sont reprises.

6.2.1 Simula67

En se basant sur le diagramme d'objets correspondant au langage étudié, dans ce cas Simula67, il est assez aisé de voir quelles classes sont utilisées par le langage. Il suffit alors de créer des individus pour chacune de ces classes. Puisque Simula67 est un langage relativement basique au niveau de l'héritage, seuls quelques concepts sont utilisés. C'est pour cela que certaines notions, telles que les interfaces, n'ont pas d'individus bien que les concepts existent dans l'ontologie principale. Pour chaque individu créé, il est possible (et même conseillé) de spécifier les différentes propriétés qui le concernent, que ce soient celles de données ou d'objet.

La Figure 23 reprend les différents individus créés pour Simula67 et la Figure 24 montre l'utilisation des propriétés d'objet utilisées pour lier l'individu "Class" aux autres individus. On y voit également deux propriétés de données définies.

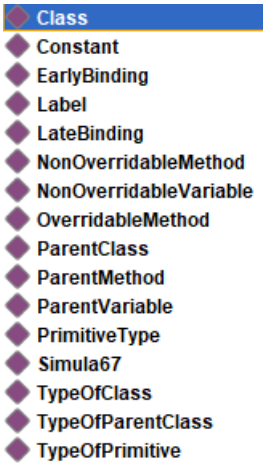

FIGURE 23 – Individus Simula67

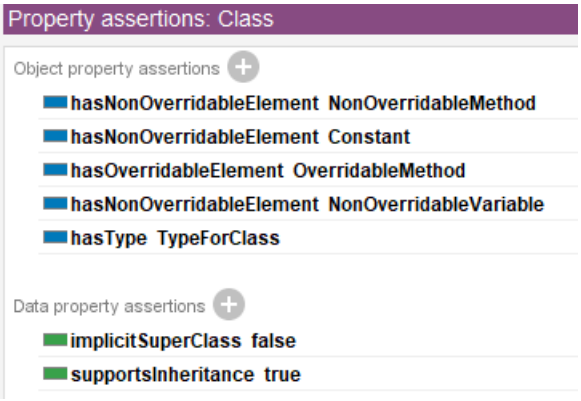

FIGURE 24 – Exemple des propriétés pour un individu

Cependant, en fonction de comment l'ontologie de base a été définie et de comment l'individu est utilisé dans les relations d'autres individus, il n'est pas toujours nécessaire de tout définir puisqu'il est possible de laisser le raisonneur inférer les divers axiomes possibles voire le type même de l'individu. La Figure 25 affiche un axiome inféré, "isTypedEntitiesSemanticsOf Simula67", sur une couleur de fond jaune. Le raisonneur a su créer cet axiome car l'individu dont il s'agit, à savoir "PrimitiveType", est utilisé dans les propriétés d'objet de l'individu "Simula67", représentant le langage.

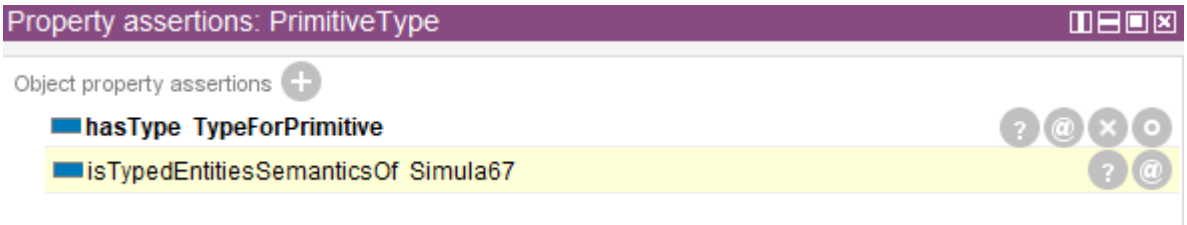

FIGURE 25 – Propriétés inférées

Il se peut que les personnes non habituées à OWL ou à Protégé trouvent que l'affichage des

individus et des propriétés n'est pas très accessible. Heureusement, il est possible d'installer divers plugins dans Protégé dont certains qui permettent d'avoir une représentation plus graphique tel que "OntoGraf" [41]. Ce plugin permet de générer un graphe comme celui affiché à la Figure 26 qui a pour avantages de faciliter la compréhension des relations entre les individus.

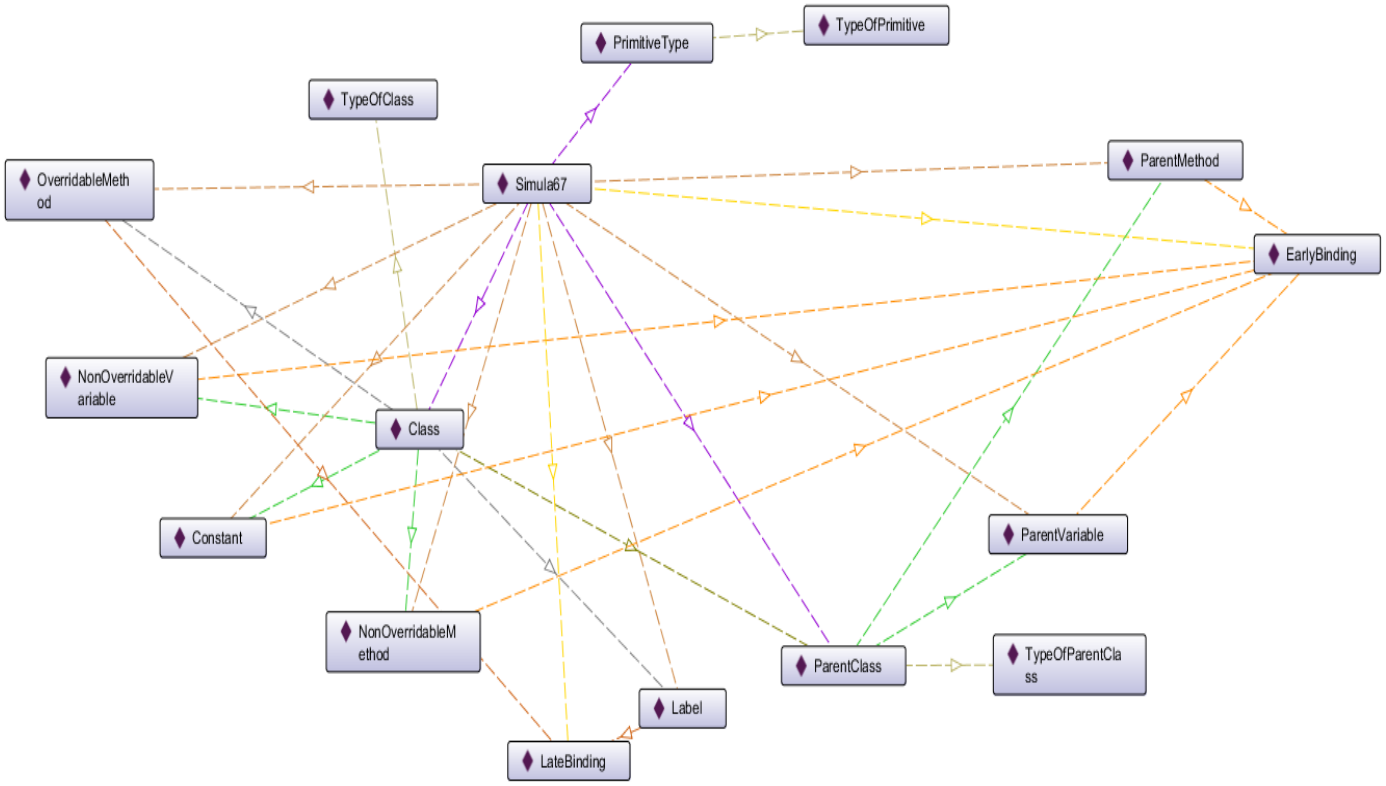

FIGURE 26 – Graphe des individus pour Simula67

6.2.2 CSharp 8

Si on applique le même processus au diagramme d'objets de C# 8, alors on obtient les individus présentés à la Figure 27. On peut observer, qu'outre les concepts propres aux langages .NET tels que les propriétés ou les indexeurs, l'ontologie se complexifie par rapport à celle présentée dans Simula67. En effet, les individus permettant de gérer les diverses notions d'interfaces ainsi que les classes abstraites ont été ajoutés. Bien entendu, chacun de ces individus est relié aux autres

41. https ://protegewiki.stanford.edu/wiki/OntoGraf

grâce aux propriétés d'objet. Si certains individus ne sont pas affichés en gras dans Protégé, c'est simplement parce que leur type n'a pas été défini manuellement mais est inférable de par leur participation dans certains axiomes.

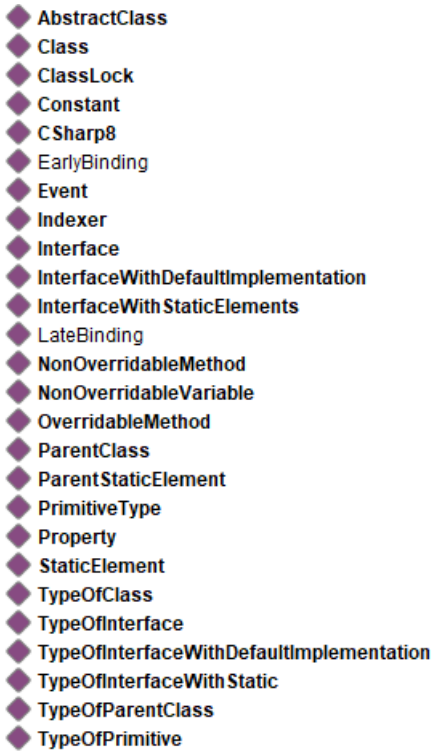

FIGURE 27 – Individus CSharp 8

Afin de représenter le fait qu'une classe a forcément une super-classe, on pourrait rajouter l'axiome représenté à la figure 28 sur la classe "Class". Le mot-clé "some" permet d'indiquer qu'il y a un au moins une super-classe. Cependant, ajouter cet axiome va forcer ce critère sur tous les individus de type "Class", y compris la classe "Object" qui est *la* classe de base et qui n'hérite de rien. Ajouter cet axiome pose donc problème, il existe plusieurs moyens d'arriver à une solution acceptable. On pourrait par exemple déclarer une autre sous-classe un peu spéciale qui représenterait la classe "Object" ou se baser sur une donnée booléenne indiquant si la classe traitée est la classe de base du langage. Mais si l'on veut être générique, alors la solution est encore de laisser les clauses tel quel.

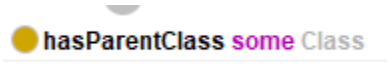

FIGURE 28 – Axiome indiquant la présence obligatoire d'une super-classe

La Figure 29 représente la version OntoGraf des individus créés. Le nombre d'individus augmentant, il devient compliqué d'avoir une vue d'ensemble nette mais cela reste visuellement plus explicite.

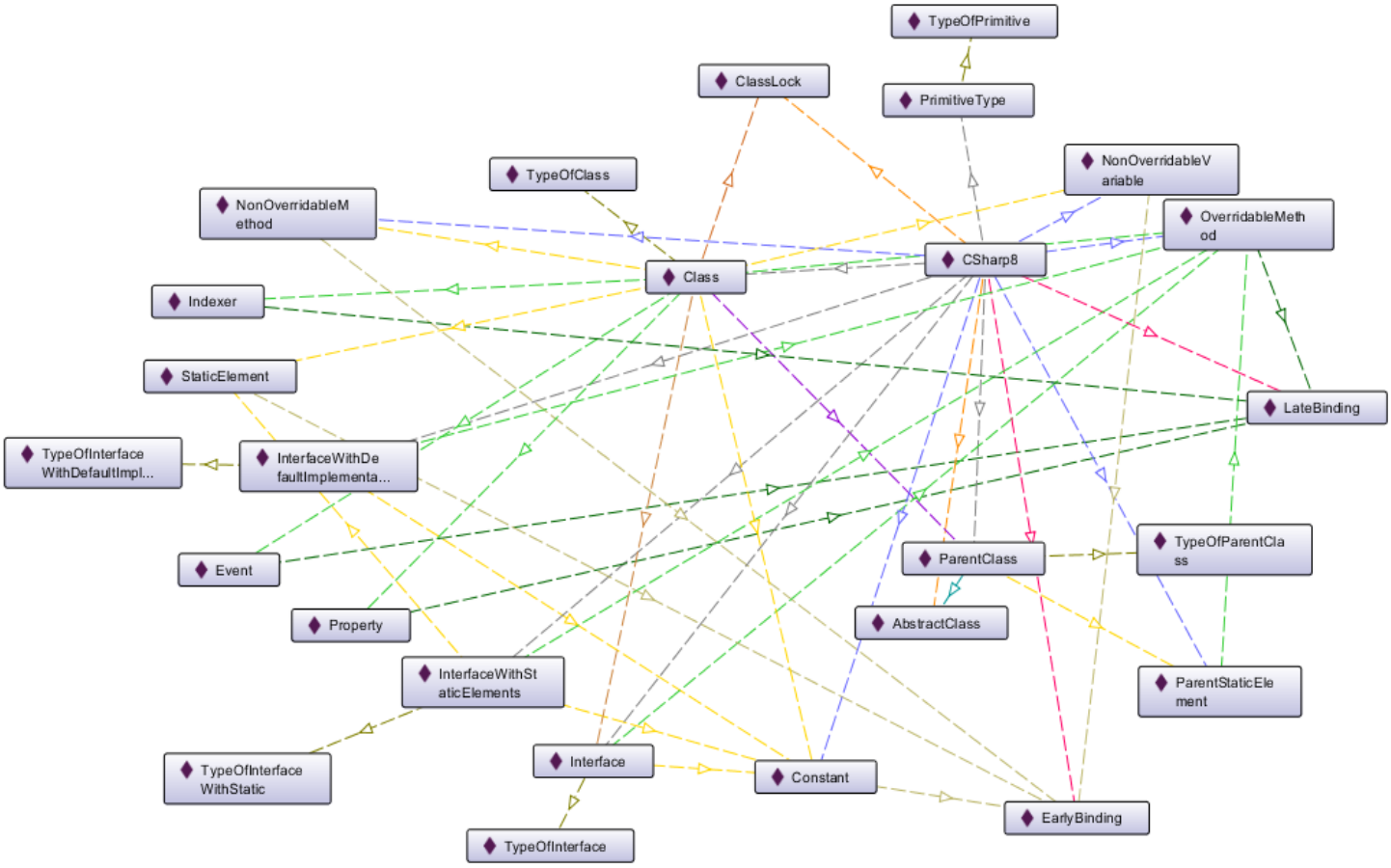

FIGURE 29 – Graphe des individus CSharp 8

6.2.3 FSharp

L'ontologie produite pour F# ressemble fortement à celle de C# 8, à la différence que certains des individus qui sont redéfinissables en C# ne le sont pas en F#. Parmi ces individus, on retrouve les indexeurs et les évènements présents dans la Figure 30.

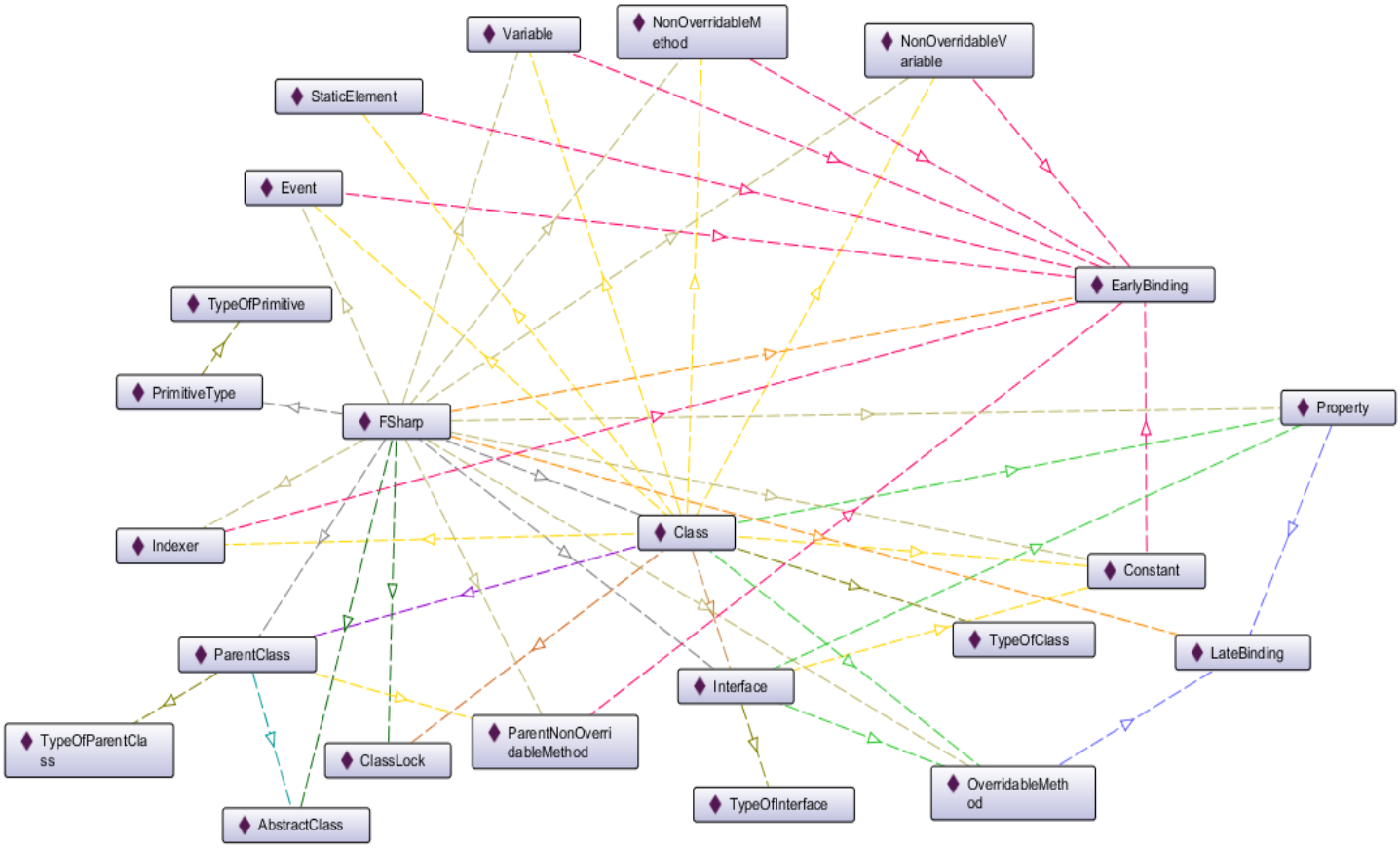

FIGURE 30 – Graphe des individus FSharp

6.2.4 Java 15

La prochaine et dernière ontologie des langages à héritage simple est celle sur Java 15. Le langage offre les mêmes fonctionnalités principales que C# 8 mais propose également divers moyens de limiter et de forcer l'héritage, grâce à un blocage total ou partiel ainsi que limité à certaines méthodes ou ciblant la classe dans son entièreté. La Figure 31 reprend les individus créés pour illustrer Java 15.

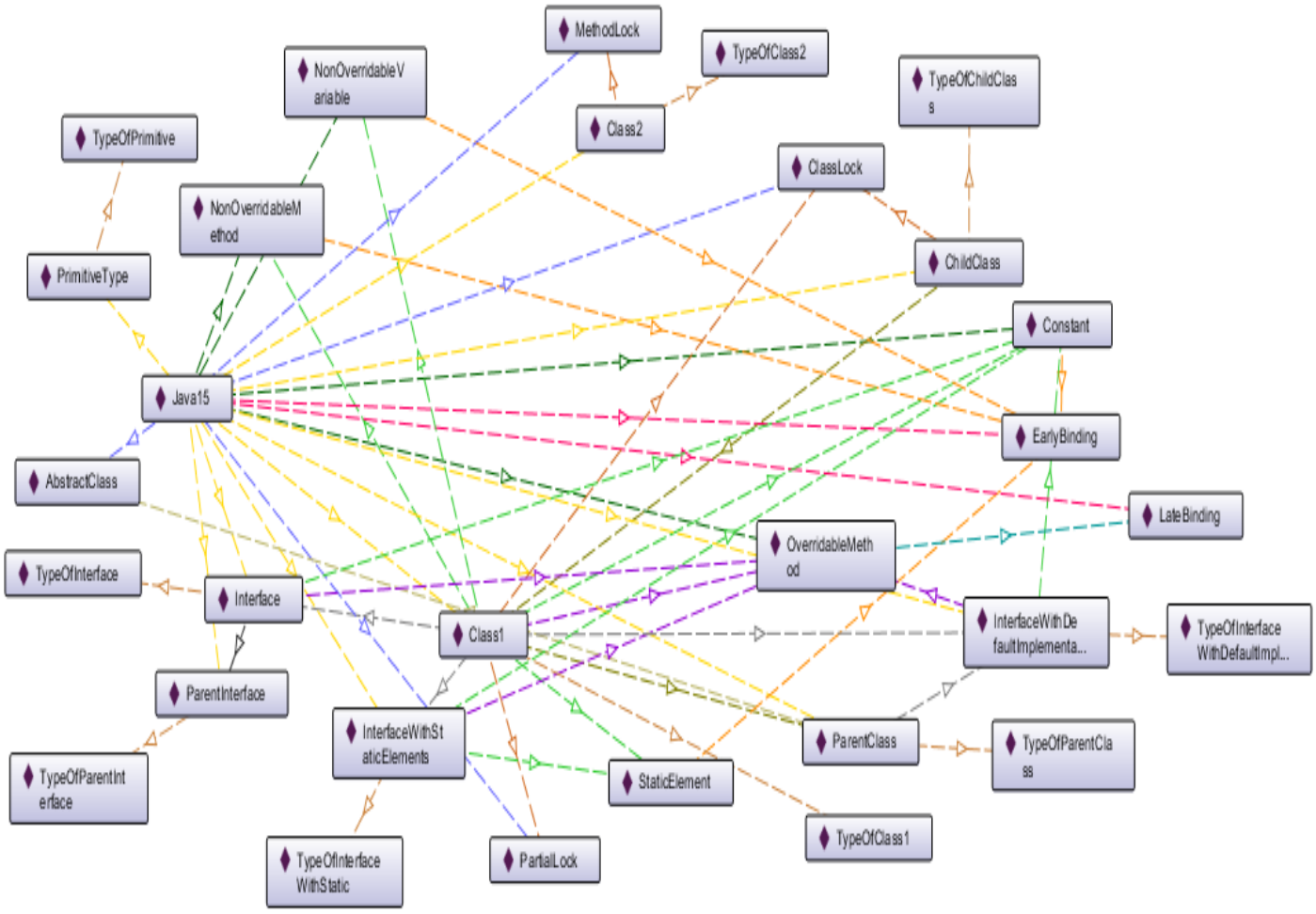

FIGURE 31 – Graphe des individus Java 15

6.2.5 C++

C++ est un langage qui supporte l'héritage multiple et n'a pas de super-classe par défaut. Tel que montré par la Figure 32, on peut avoir plusieurs super-classes pour une seule classe enfant. Outre cette différence, on observe que les concepts utilisés restent relativement les mêmes bien que les interfaces soient moins développées que dans les précédents langages et qu'il n'y ait pas de moyen pour limiter l'héritage à quelques classes uniquement, même si on peut le faire au niveau de la classe ou de la méthode.

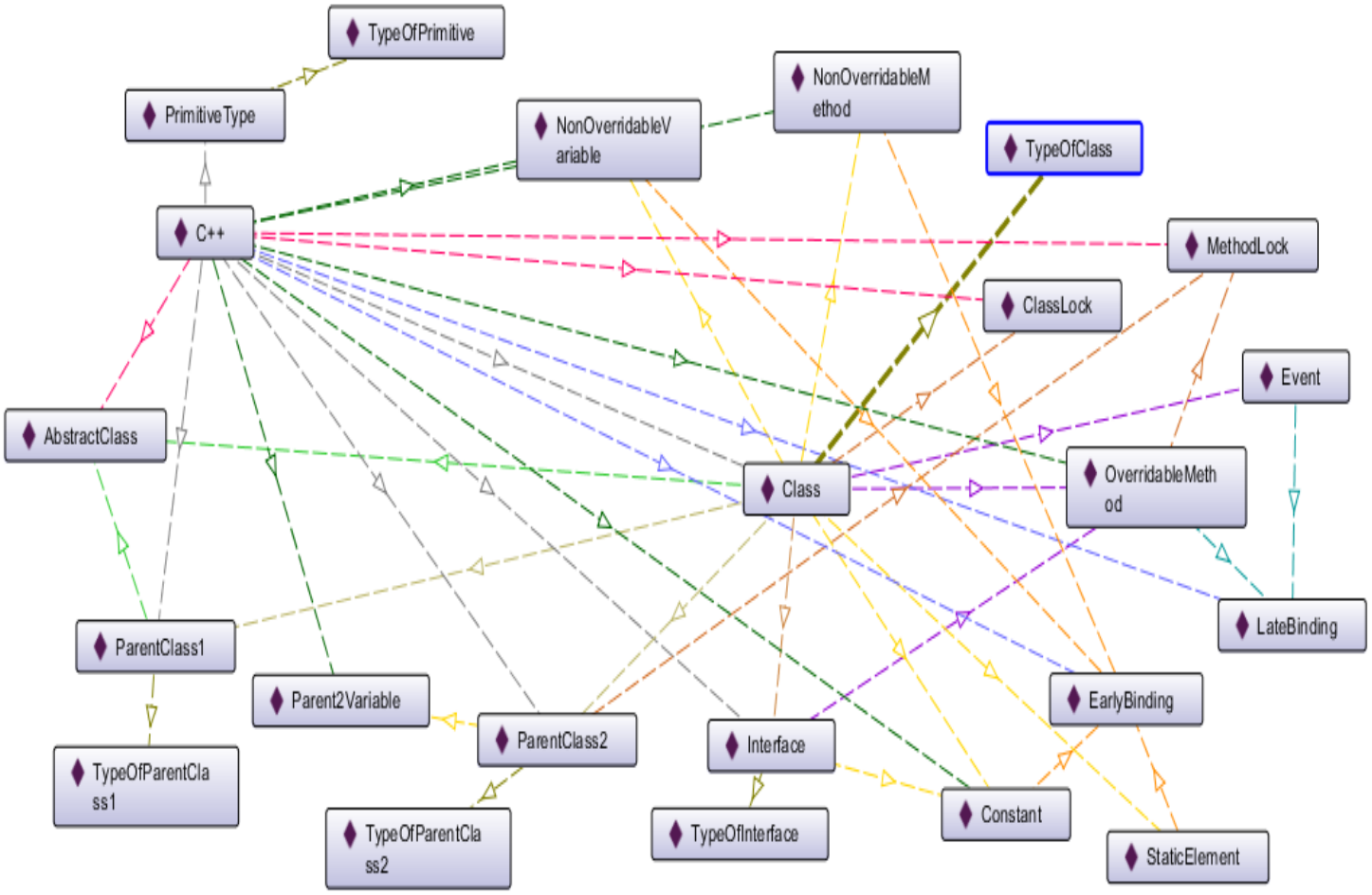

FIGURE 32 – Graphe des individus C++

6.2.6 OCaml

OCaml, second langage usant de l'héritage multiple, est bien plus simple à représenter puisque de nombreux concepts sont absents ou se comportent d'une manière différente. On note par exemple l'absence de procédés permettant de limiter ou de forcer l'héritage ou encore le fait qu'une classe ne peut toujours être liée qu'à une seule interface. Ce dernier point entraîne d'ailleurs une modification dans la définition du concept de classe telle qu'affichée dans la Figure 33. De cette manière, une classe aura toujours exactement une seule interface.

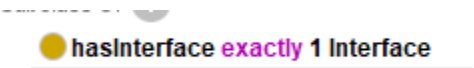

FIGURE 33 – Limitation à une interface

Les différents individus composant l'ontologie pour OCaml sont présents dans le graphe à la Figure 34.

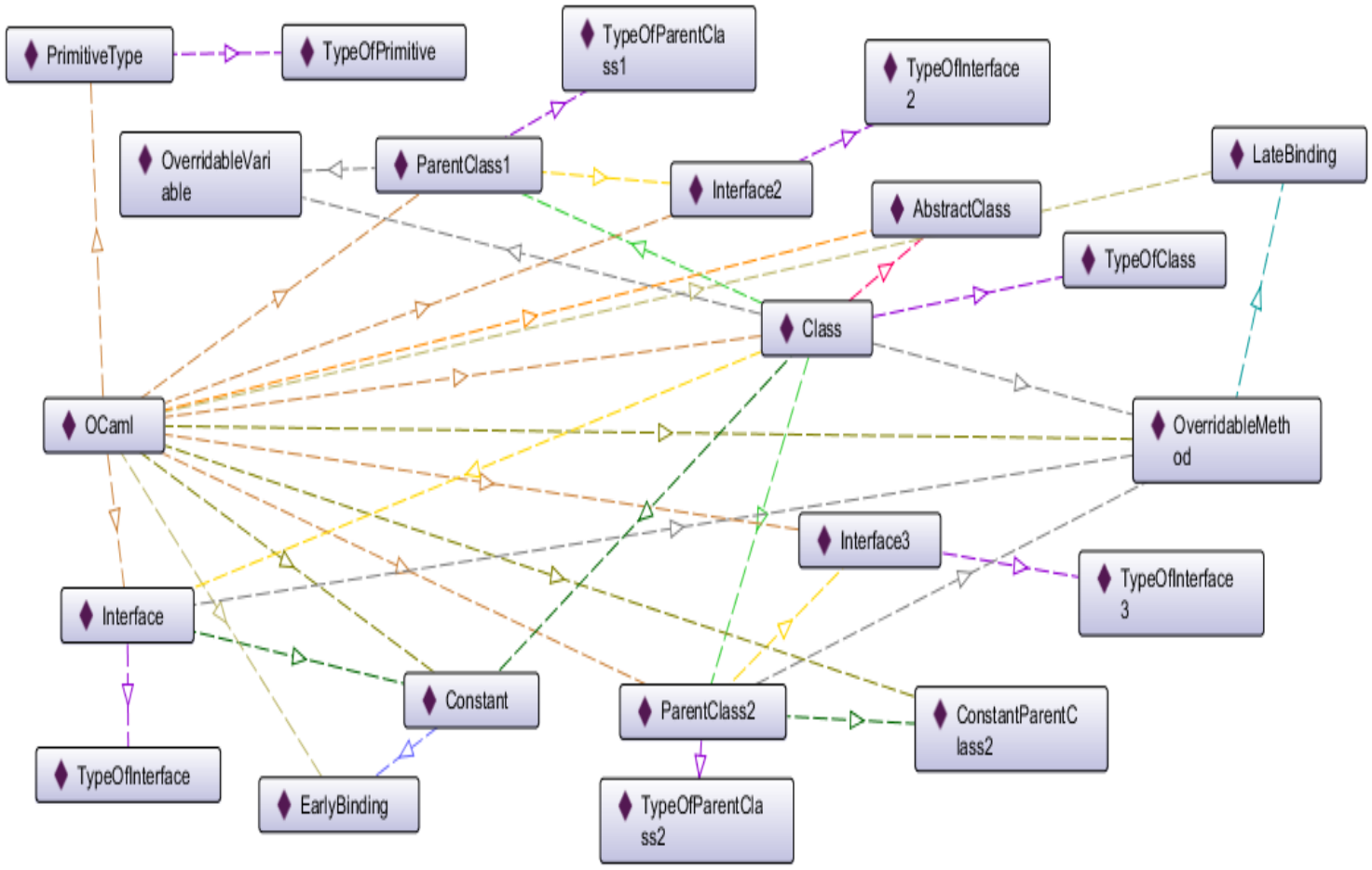

FIGURE 34 – Graphe des individus OCaml

6.2.7 Haskell

Tout comme l'ontologie d'OCaml, celle d'Haskell est relativement simple puisque le langage ne supporte pas l'héritage tel qu'un langage orienté objet le ferait. Les différents individus produits permettent néanmoins de simuler les différents concepts qui ont été définis dans l'ontologie principale. C'est pourquoi qu'on obtient par exemple un individu représentant une classe abstraite bien que, comme expliqué dans le chapitre 4.3.1, les classes abstraites n'existent pas en tant que telles dans Haskell. On observe qu'il est possible d'imiter en partie le comportement des langages orientés objet vis-à-vis des interfaces mais qu'on est vite bloqué lorsqu'on souhaite simuler l'héritage en lui-même. C'est ainsi que l'on arrive au graphe 35 reprenant les divers individus créés.

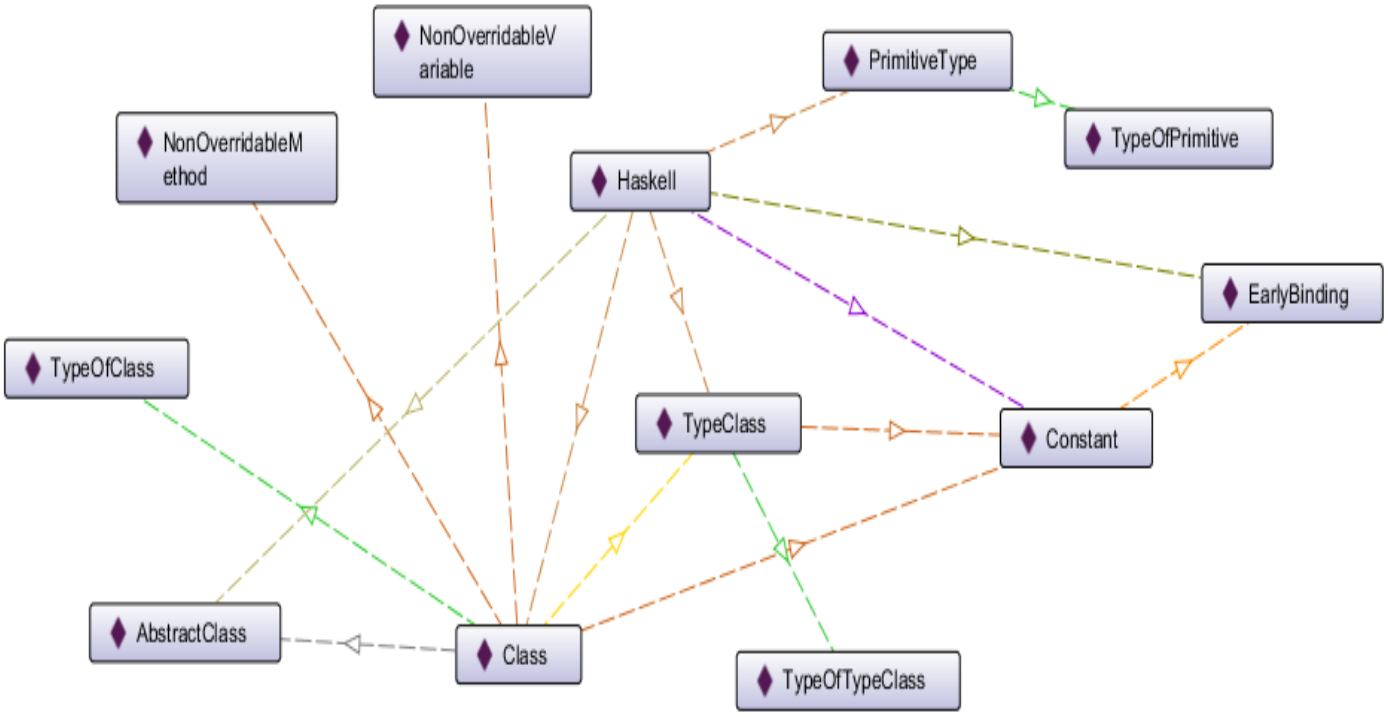

FIGURE 35 – Graphe des individus Haskell

7 Limites et travaux futurs

Dans cette section seront abordés les limites et problèmes rencontrés durant la réalisation de ce travail. Quelques pistes pour de futurs travaux seront également abordées.

7.1 Limites

La création d'ontologies étant un sujet vaste et non complètement maîtrisé, il est tout à fait possible que les ontologies créées ne soient pas optimales voire contiennent des erreurs, bien que le raisonneur élimine déjà la possibilité d'avoir des erreurs de cohérence. Avoir une meilleure connaissance et maîtrise d'OWL et des ontologies en général permettrait de limiter les erreurs potentielles. De plus, le méta-modèle ainsi que l'ontologie principale sont des représentations et chacun peut avoir sa propre représentation. Il est donc tout à fait possible qu'un autre travail sur la même problématique arrive à des ontologies différentes dans leur composition. Néanmoins, le résultat devrait rester globalement semblable.

La méconnaissance de certains langages a pu engendrer des imprécisions ou des erreurs de compréhension, bien qu'elles devraient être minimes voire absentes puisque la documentation officielle du langage a été utilisé, quand cela était possible. Cependant, certains langages étant plus vieux et légèrement tombés en désuétude, il fut compliqué de trouver de la documentation exhaustive ou des exemples récents. C'est notamment le cas pour Simula67 qui n'est plus activement utilisé de nos jours. Cela complique le travail de recherche mais ne devrait pas avoir engendré d'erreurs. Discuter avec des professionnels ayant utilisé ces langages pourraient également fournir de nouvelles pistes de réflexion.

Afin que ce travail reste à une échelle réalisable, les langages de programmation étudiés ont été limités. Il serait intéressant de réaliser ce même travail avec d'autres langages populaires tel que Python ou Ruby. On pourrait également s'intéresser à des langages qui ont adopté le paradigme orienté objet sur le tard comme PHP qui le supporte depuis la 5ème version du langage. D'autres langages également pertinents seraient Smalltalk et Eiffel. Smalltalk a énormément influencé les langages orienté objets et a pour particularité que *tout* est objet. Eiffel[42], quant à lui, a été conceptualisé par Bertrand Meyer qui est une personnalité importante dans le monde de l'orienté objet, il fut l'un des premiers à rédiger un livre sur ce paradigme, définissant bon nombre de concepts.

7.2 Travaux futurs

Afin de pouvoir tirer parti du travail effectué, il pourrait être intéressant de créer de nouvelles classes dans l'ontologie principale et d'appliquer le raisonneur dessus afin de voir quels langages répondent aux critères de ces classes. Ceci dit, pour pouvoir réaliser cela de manière efficace et centralisée, il faudrait avoir toutes les ontologies réunies en une seule. Mais cela permettrait de pousser l'analyse et d'augmenter l'intérêt d'avoir réalisé une ontologie. Un exemple d'une telle classe était déjà présent dans le mémoire, à savoir les langages qui ont des classes abstraites, mais il existe plusieurs autres critères qui pourraient être intéressants tels que savoir quels langages n'ont que des éléments redéfinissables, n'ont pas de super-classe par défaut, etc.

42. https ://fr.wikipedia.org/wiki/Eiffel_(langage)

Outre le fait d'étudier des langages supplémentaires, une amélioration possible serait d'étudier d'autres caractéristiques liées à l'héritage. Une caractéristique liée à l'héritage et non étudiée dans ce travail est l'héritage des types génériques. Les types génériques, bien que non supportés par tous les langages, permettent d'apporter une plus grande flexibilité et réutilisabilité. Ils sont présents dans nombre de langages modernes et se pencher sur l'héritage de ces types pourrait être fortement intéressant.

Maintenant que la sémantique de l'héritage dans ces langages a été définie, il peut être intéressant de s'intéresser à comment l'héritage est implémenté, de manière pratique. Quels sont les différents compilateurs pour un même langage, quels sont les points communs et les différences ? Est-ce que l'héritage n'est rien de plus qu'une concaténation des classes, d'un point de vue physique ? Toutes ces questions pourraient être répondues en suivant la méthodologie utilisée pour ce mémoire.

Il a été choisi de se limiter à l'héritage mais le même travail pourrait être appliqué à d'autres domaines. Il a été fait référence à quelques reprises de la programmation fonctionnelle, il serait peut-être intéressant de la comparer avec la programmation objet. Ou alors se pencher plus amplement sur les langages statiques et les langages dynamiques, et ce peu importe le paradigme de prédilection des langages étudiés.

8 Conclusion

Les langages de programmation sont nombreux et peuvent être radicalement différents les uns des autres. Néanmoins, ils peuvent, pour la plupart, être classés en fonction des paradigmes de programmation qu'ils supportent. Un paradigme populaire de nos jours est le paradigme orienté objet et les langages utilisant ce paradigme partagent généralement plusieurs concepts communs. Parmi ces concepts, on retrouve l'héritage, concept de base présent dans les langages orientés objet mais extrêmement important.

Les ontologies permettent de représenter des bases de connaissance de manière formelle, de pouvoir raisonner dessus et de les questionner. Analyser la manière dont l'héritage fonctionne dans les langages de programmation et représenter cela sous forme d'ontologie permet d'avoir une vue d'ensemble facilement réutilisable et distribuable.

Ce document a commencé par un travail de recherche afin de poser les bases des différents éléments abordés tout au long de ce mémoire et de comprendre comment les ontologies fonctionnent. L'état de l'art qui suivit a permis de relever les bonnes pratiques lors de la conception d'ontologies et surtout de mettre en avant ce qui existait déjà. Afin de répondre à la question de recherche, qui était de savoir comment représenter le concept d'héritage selon une perspective ontologique, il a fallu tout d'abord analyser l'héritage dans chacun des langages étudiés. Une fois ceci fait, la conception d'un méta-modèle permit d'avoir une vue d'ensemble des notions à aborder tout en fournissant une représentation visuelle plus accessible pour les personnes n'étant pas habituées aux ontologies. Et c'est à partir de ce méta-modèle que l'ontologie principale a pu être rédigée.

C'est lors de l'analyse des divers langages que les grandes caractéristiques à étudier ont été mises en avant. Pour rappel, il s'agissait des principes de base de l'héritage, de la manière dont les éléments étaient redéfinis et dissimulés, des moyens pour limiter ou forcer la propagation de l'héritage et enfin la notion d'interface. Étudier les langages sous ces divers aspects permet d'avoir une compréhension suffisante de comment l'héritage a été pensé et défini. Les langages étudiés ont été séparés en trois catégories, en fonction de s'ils ne supportent que l'héritage simple, supportent l'héritage multiple ou ne supportent pas du tout l'héritage. Même si ce travail permet de comparer tous les langages au point de vue de l'héritage, cette catégorisation facilite la comparaison entre des langages identiques.

La création du méta-modèle a mis en avant les liens entre les différentes notions analysées précédemment. Le méta-modèle permet d'entrevoir ce à quoi ressemblera l'ontologie principale. Les concepts présents dans le méta-modèle n'étaient pas tous explicités lors de l'étape de l'analyse mais la conception de celui-ci a permis de mettre au clair tout ce que l'ontologie devra utiliser.

L'objectif initial de ce mémoire était donc de réaliser une ontologie représentant le concept de l'héritage. Finalement, plusieurs ontologies ont été produites, une principale, permettant de définir les grands concepts analysés auparavant et déjà représentés dans le méta-modèle, et plusieurs ontologies gravitant autour de l'ontologie principale. Ces dernières ont pour particularité d'être chacune dédiée à un langage (ou à une version de langage si cela s'avérait pertinent). Une telle découpe permet une meilleure clarté tout en tirant avantage de la facilité d'importer des ontologies.

L'ontologie principale créée peut être bien entendu perfectionnée mais elle propose déjà une représentation de l'héritage au sein de plusieurs langages et met en avant les divers concepts utilisés ainsi que les relations qui les lient.

Références

[1] C++ documentation. https ://learn.microsoft.com/fr-fr/cpp/cpp/. Accessed : 2022-05-01.

[2] C# documentation. https ://learn.microsoft.com/en-us/dotnet/csharp/. Accessed : 2021-11-09.

[3] Dictionnaire Le Robert. https ://dictionnaire.lerobert.com/definition/ontologie. Accédé : 2022-03-15.

[4] Haskell - Equality documentation). https ://hackage.haskell.org/package/base-4.18.0.0/docs/Data-Eq.html. Accessed : 2023-02-10.

[5] The history of C#. https ://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history. Accessed : 2023-02-13.

[6] Interfaces (F#). https ://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/interfaces. Accessed : 2023-02-12.

[7] Java documentation. https ://docs.oracle.com/en/java/. Accessed : 2021-11-10.

[8] Le principe de substitution de Liskov. https ://en.wikipedia.org/wiki/Liskov$_s ubstitution_p rinciple. Access$
$2023 - 02 - 13$.

[9] Multiple Inheritance (C++). https ://learn.microsoft.com/en-us/cpp/cpp/multiple-base-classes ?view=msvc-170. Accessed : 2023-02-12.

[10] Object Expressions (F#). https ://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/object-expressions. Accessed : 2023-02-12.

[11] Objects in OCaml. https ://v2.ocaml.org/manual/objectexamples.html. Accessed : 2023-02-10.

[12] OCaml documentation. https ://ocaml.org/docs. Accessed : 2022-05-01.

[13] Releases (OCaml). https ://ocaml.org/releases/3.12.1. Accessed : 2023-02-10.

[14] Scala documentation. https ://docs.scala-lang.org/. Accessed : 2022-05-01.

[15] Sealed Classes (Scala). https ://docs.scala-lang.org/tour/pattern-matching.htmlsealed-classes. Accessed : 2023-02-12.

[16] Simple Inheritance (C++). https ://learn.microsoft.com/en-us/cpp/cpp/single-inheritance ?view=msvc-170. Accessed : 2023-02-12.

[17] TypeClasses (Haskell). https ://serokell.io/blog/haskell-typeclasses. Accessed : 2023-02-10.

[18] Izzeddin A. O. Abuhassan and A. Almashaykhi. Domain Ontology for Programming Languages. 2012.

[19] Awny Alnusair and Tian Zhao. Component search and reuse : An ontology-based approach. In *2010 IEEE International Conference on Information Reuse Integration*, pages 258–261, 2010.

[20] Mattia Atzeni and Maurizio Atzori. CodeOntology : RDF-ization of Source Code. In Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web – ISWC 2017*, pages 20–28, Cham, 2017. Springer International Publishing.

[21] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics as ontology languages for the semantic web. In *Mechanizing mathematical reasoning*, pages 228–248. Springer, 2005.

[22] Jon Barwise. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 5–46. Elsevier, 1977.

[23] Daniel G. Bobrow and Terry Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1) :3–46, 1977.

[24] Willem Nico Borst. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, University of Twente, Netherlands, sep 1997.

[25] Camila Zacché de Aguiar, Ricardo de Almeida Falbo, and Vítor E. Silva Souza. OOC-O : A Reference Ontology on Object-Oriented Code. In Alberto H. F. Laender, Barbara Pernici, Ee-Peng Lim, and José Palazzo M. de Oliveira, editors, *Conceptual Modeling*, pages 13–27, Cham, 2019. Springer International Publishing.

[26] Giuseppe De Giacomo and Maurizio Lenzerini. TBox and ABox reasoning in expressive description logics. *KR*, 96(316-327) :10, 1996.

[27] Baboucar Diatta, Adrien Basse, and Samuel Ouya. PasOnto : Ontology for Learning Pascal Programming Language. In *2019 IEEE Global Engineering Education Conference (EDUCON)*, pages 749–754, 2019.

[28] Claudiu Epure and Adrian Iftene. Semantic Analysis of Source Code in Object Oriented Programming . A Case Study for C #. 2016.

[29] Alyce Faulstich-Brady. A taxonomy of inheritance semantics. In *Proceedings of 1993 IEEE 7th International Workshop on Software Specification and Design*, pages 194–203. IEEE, 1993.

[30] Richard Fikes, Patrick Hayes, and Ian Horrocks. OWL-QL—a language for deductive query answering on the Semantic Web. *Journal of Web Semantics*, 2(1) :19–29, 2004.

[31] A. Gómez-Pérez. Développement récents en matière de conception, de maintenance et d'utilisation des ontologies. *Terminologies nouvelles*, 19 :9–20, 1999. Ontology Engineering Group OEG.

[32] Object Management Group. Ontology Definition Metamodel. https ://www.omg.org/spec/ODM/1.1. Accessed : 2023-04-22.

[33] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5 :199–220, 1993.

[34] Giancarlo Guizzardi and Gerd Wagner. A Unified Foundational Ontology and some Applications of it in Business Modeling. pages 129–143, 01 2004.

[35] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2) :109–138, mar 1996.

[36] Diana Kalibatiene and Olegas Vasilecas. Survey on ontology languages. In *International Conference on Business Informatics Research*, pages 124–141. Springer, 2011.

[37] Aggeliki Kouneli, Georgia Solomou, Christos Pierrakeas, and Achilles Kameas. Modeling the Knowledge Domain of the Java Programming Language as an Ontology. In Elvira Popescu, Qing Li, Ralf Klamma, Howard Leung, and Marcus Specht, editors, *Advances in Web-Based Learning - ICWL 2012*, pages 152–159, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[38] Andrea Marchetti, Francesco Ronzano, Maurizio Tesconi, and Salvatore Minutoli. Formalizing Knowledge by Ontologies : OWL and KIF. 05 2008.

[39] John McCarthy. Circumscription—A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1) :27–39, 1980. Special Issue on Non-Monotonic Logic.

[40] B. Meyer. The many faces of inheritance : A taxonomy of taxonomy. *Computer*, 29(05) :105–108, may 1996.

[41] Marvin Minsky. A framework for representing knowledge, 1974.

[42] Mark A Musen et al. Ontology-oriented design and programming. *Knowledge engineering and agent technology*, 52 :3–16, 2000.

[43] Daniele Nardi, Ronald J Brachman, et al. An introduction to description logics. *Description logic handbook*, 1 :40, 2003.

[44] Robert Neches, Richard E. Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William R. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3) :36, Sep. 1991.

[45] Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA, 1959.

[46] N. Noy and Deborah Mcguinness. Ontology Development 101 : A Guide to Creating Your First Ontology. *Knowledge Systems Laboratory*, 32, 01 2001.

[47] Meilir Page-Jones. *Fundamentals of object-oriented design in UML*. Pearson Education India, 2000.

[48] Ramangouda Patil, Richard Fikes, Peter Patel-schneider, Don Mckay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA knowledge sharing effort : Progress report. 07 1998.

[49] Christos Pierrakeas, Georgia Solomou, and Achilles Kameas. An Ontology-Based Approach in Learning Programming Languages. In *2012 16th Panhellenic Conference on Informatics*, pages 393–398, 2012.

[50] Robert J Pooley. *An introduction to programming in SIMULA*. Blackwell Scientific Publications Oxford, 1987.

[51] M Ross Quillian. Word concepts : A theory and simulation of some basic semantic capabilities. *Behavioral science*, 12(5) :410–430, 1967.

[52] Guus Schreiber. OWL : the Web Ontology Language. https ://www.cs.vu.nl/ guus/talks/04-owl-brisbane/all.htm. Accessed : 2023-02-18.

[53] Stewart Shapiro. Classical Logic II : Higher-Order Logic. *The Blackwell guide to philosophical logic*, pages 33–54, 2017.

[54] Barry Smith and Christopher Welty. FOIS Introduction : Ontology—towards a New Synthesis. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, FOIS '01, page .3–.9, New York, NY, USA, 2001. Association for Computing Machinery.

[55] Sergey Sosnovsky and Tatiana Gavrilova. Development of educational ontology for C-programming. *International Journal on Information Theories Applications*, 13 :303–308, 01 2006.

[56] Antero Taivalsaari. On the Notion of Inheritance. *ACM Comput. Surv.*, 28(3) :438–479, sep 1996.

[57] Ophélie Tremblay and Alain Polguère. Une ontologie linguistique au service de la didactique du lexique. In *SHS Web of Conferences*, volume 8, pages 1173–1188. EDP Sciences, 2014.

[58] Sylvain Turcotte, Joanne Otis, and Louise Gaudreau. Les objets d'enseignement-apprentissage : éléments d'illustration de l'inclusion de l'éducation à la santé en éducation physique. *Staps*, 75, 01 2007.

[59] W3C. OWL Web Ontology Language - Overview. https ://www.w3.org/TR/2004/REC-owl-features-20040210/. Accessed : 2023-04-22.

9 Annexes

9.1 Code de l'ontologie principale

Le code correspondant à l'ontologie principale se trouve dans un fichier dont l'extension est "owl" et suit donc la syntaxe Manchester. Puisque le code en question fait plus de 700 lignes, il est découpé en plusieurs images reprises ci-dessous.

```
Prefix: : <http://www.semanticweb.org/utilisateur/ontologies/2023/0/untitled-ontology-86#>
Prefix: dc: <http://purl.org/dc/elements/1.1/>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>



Ontology: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology>


Datatype: xsd:boolean


ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBinding>

    SubPropertyOf:
        owl:topObjectProperty

    InverseOf:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isBindingOf>


ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBindingSemantics>

    SubPropertyOf:
        owl:topObjectProperty

    Domain:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>

    Range:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>

    InverseOf:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isBindingSemanticsOf>


ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasEarlyBinding>

    SubPropertyOf:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBinding>

    Domain:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>

    Range:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#EarlyBinding>

    InverseOf:
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isEarlyBindingOf>
```

FIGURE 36 – Partie 1

```
55
56  ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElement>
57
58      SubPropertyOf:
59          owl:topObjectProperty
60
61      Domain:
62          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
63
64      Range:
65          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>
66
67      InverseOf:
68          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isElementOf>
69
70
71  ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElementSemantics>
72
73      SubPropertyOf:
74          owl:topObjectProperty
75
76      Domain:
77          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
78
79      Range:
80          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>
81
82      InverseOf:
83          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isElementSemanticsOf>
84
85
86  ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasForceFurtherInheritance>
87
88      InverseOf:
89          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isForceFurtherInheritance>
90
91
92  ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagation>
93
94      InverseOf:
95          <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInheritancePropagationOf>
96
97
98  ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagationSemantics>
99
100     SubPropertyOf:
101         owl:topObjectProperty
102
103     Domain:
104         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
105
106     Range:
107         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InheritancePropagation>
108
109     InverseOf:
110         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInheritancePropagationSemanticsOf>
111
```

FIGURE 37 – Partie 2

```
111
112
113 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInterface>
114
115     SubPropertyOf:
116         owl:topObjectProperty
117
118     Domain:
119         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>
120
121     Range:
122         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
123
124     InverseOf:
125         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInterfaceOf>
126
127
128 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasLateBinding>
129
130     SubPropertyOf:
131         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBinding>
132
133     Domain:
134         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
135
136     Range:
137         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LateBinding>
138
139     InverseOf:
140         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isLateBindingOf>
141
142
143 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasNonOverridableElement>
144
145     SubPropertyOf:
146         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElement>
147
148     Domain:
149         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class> or
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
150
151     Range:
152         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
153
154     InverseOf:
155         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isNonOverridableElementOf>
156
```

FIGURE 38 – Partie 3

```
157
158 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasOverridableElement>
159
160     SubPropertyOf:
161         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElement>
162
163     Domain:
164         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class> or
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
165
166     Range:
167         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
168
169     InverseOf:
170         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isOverridableElementOf>
171
172
173 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasParentClass>
174
175     SubPropertyOf:
176         owl:topObjectProperty
177
178     Domain:
179         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>
180
181     Range:
182         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>
183
184     InverseOf:
185         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isParentClassOf>
186
187
188 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasParentInterface>
189
190     SubPropertyOf:
191         owl:topObjectProperty
192
193     Domain:
194         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
195
196     Range:
197         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
198
199     InverseOf:
200         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isParentInterfaceOf>
201
202
203 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasPreventFurtherInheritance>
204
205     Domain:
206         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>
207
208     Range:
209         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PreventFurtherInheritance>
210
211     InverseOf:
212         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isPreventFurtherInheritanceOf>
213
```

FIGURE 39 – Partie 4

```
213
214
215 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasType>
216
217     SubPropertyOf:
218         owl:topObjectProperty
219
220     Domain:
221         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
222
223     Range:
224         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Type>
225
226     InverseOf:
227         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isTypeOf>
228
229
230 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasTypedEntitiesSemantics>
231
232     SubPropertyOf:
233         owl:topObjectProperty
234
235     Domain:
236         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
237
238     Range:
239         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
240
241     InverseOf:
242         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isTypedEntitiesSemanticsOf>
243
244
245 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isBindingOf>
246
247     SubPropertyOf:
248         owl:topObjectProperty
249
250     InverseOf:
251         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBinding>
252
253
254 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isBindingSemanticsOf>
255
256     InverseOf:
257         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBindingSemantics>
258
259
260 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isEarlyBindingOf>
261
262     SubPropertyOf:
263         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isBindingOf>
264
265     InverseOf:
266         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasEarlyBinding>
267
```

FIGURE 40 – Partie 5

```
268
269 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isElementOf>
270
271     SubPropertyOf:
272         owl:topObjectProperty
273
274     InverseOf:
275         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElement>
276
277
278 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isElementSemanticsOf>
279
280     InverseOf:
281         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElementSemantics>
282
283
284 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isForceFurtherInheritance>
285
286     InverseOf:
287         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasForceFurtherInheritance>
288
289
290 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInheritancePropagationOf>
291
292     SubPropertyOf:
293         owl:topObjectProperty
294
295     InverseOf:
296         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagation>
297
298
299 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInheritancePropagationSemanticsOf>
300
301     SubPropertyOf:
302         owl:topObjectProperty
303
304     InverseOf:
305         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagationSemantics>
306
307
308 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInterfaceOf>
309
310     InverseOf:
311         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInterface>
312
313
314 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isLateBindingOf>
315
316     SubPropertyOf:
317         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isBindingOf>
318
319     InverseOf:
320         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasLateBinding>
321
```

FIGURE 41 – Partie 6

```
321
322
323 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isNonOverridableElementOf>
324
325     SubPropertyOf:
326         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isElementOf>
327
328     InverseOf:
329         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasNonOverridableElement>
330
331
332 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isOverridableElementOf>
333
334     SubPropertyOf:
335         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isElementOf>
336
337     InverseOf:
338         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasOverridableElement>
339
340
341 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isParentClassOf>
342
343     InverseOf:
344         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasParentClass>
345
346
347 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isParentInterfaceOf>
348
349     SubPropertyOf:
350         owl:topObjectProperty
351
352     InverseOf:
353         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasParentInterface>
354
355
356 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isPreventFurtherInheritanceOf>
357
358     InverseOf:
359         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasPreventFurtherInheritance>
360
361
362 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isTypeOf>
363
364     SubPropertyOf:
365         owl:topObjectProperty
366
367     InverseOf:
368         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasType>
369
370
371 ObjectProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isTypedEntitiesSemanticsOf>
372
373     SubPropertyOf:
374         owl:topObjectProperty
375
376     InverseOf:
377         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasTypedEntitiesSemantics>
378
```

FIGURE 42 – Partie 7

```
378
379
380 ObjectProperty: owl:topObjectProperty
381
382
383 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#explicitOverride>
384
385     SubPropertyOf:
386         owl:topDataProperty
387
388
389 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#implicitSuperClass>
390
391     SubPropertyOf:
392         owl:topDataProperty
393
394
395 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isHideable>
396
397     Range:
398         xsd:boolean
399
400
401 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#languageName>
402
403
404 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#languageVersion>
405
406     SubPropertyOf:
407         owl:topDataProperty
408
409
410 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#supportsInheritance>
411
412     SubPropertyOf:
413         owl:topDataProperty
414
415
416 DataProperty: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#variableSpecification>
417
418
419 DataProperty: owl:topDataProperty
420
421
422 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#AbstracClass>
423
424     SubClassOf:
425         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#ForceFurtherInheritance>
426
427
428 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>
429
430     DisjointWith:
431         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>
432
```

FIGURE 43 – Partie 8

```
432
433
434 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>
435
436     EquivalentTo:
437         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
438          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElement> min 0
              <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>)
439          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInterface> min 0
              <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>)
440          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasParentClass> min 0
              <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>)
441          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isParentClassOf> min 0
              <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>)
442          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagation> max 1
              <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InheritancePropagation>)
443
444
445 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#ClassLock>
446
447     SubClassOf:
448         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#FullLock>
449
450     DisjointWith:
451         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#MethodLock>
452
453
454 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Constant>
455
456     SubClassOf:
457         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
458
459
460 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#DynamicallyTypedLanguage>
461
462     SubClassOf:
463         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
464
465     DisjointWith:
466         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#StaticallyTypedLanguage>
467
468
469 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#EarlyBinding>
470
471     SubClassOf:
472         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>
473
474     DisjointWith:
475         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LateBinding>
476
477
478 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>
479
480     DisjointWith:
481         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>
482
```

FIGURE 44 – Partie 9

```
482
483
484 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Event>
485
486     SubClassOf:
487         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
488
489
490 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#ForceFurtherInheritance>
491
492     SubClassOf:
493         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InheritancePropagation>
494
495     DisjointWith:
496         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PreventFurtherInheritance>
497
498
499 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#FullLock>
500
501     SubClassOf:
502         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PreventFurtherInheritance>
503
504     DisjointWith:
505         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PartialLock>
506
507
508 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Indexer>
509
510     SubClassOf:
511         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
512
513
514 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InheritancePropagation>
515
516     SubClassOf:
517         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isInheritancePropagationOf> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>
518
519
520 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
521
522     SubClassOf:
523         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>,
524         (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasNonOverridableElement> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Constant>)
525          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasOverridableElement> some
              <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>),
526         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasParentInterface> min 0
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>,
527         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isParentInterfaceOf> min 0
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
528
```

FIGURE 45 – Partie 10

```
528
529
530 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InterfaceWithDefaultImplementation>
531
532     SubClassOf:
533         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>
534
535     DisjointWith:
536         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InterfaceWithStaticElements>
537
538
539 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InterfaceWithStaticElements>
540
541     SubClassOf:
542         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Interface>,
543         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasNonOverridableElement> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#StaticElement>
544
545     DisjointWith:
546         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InterfaceWithDefaultImplementation>
547
548
549 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Label>
550
551     SubClassOf:
552         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
553
554
555 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
556
557     SubClassOf:
558         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasBindingSemantics> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>,
559         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasElementSemantics> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>,
560         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagationSemantics> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InheritancePropagation>,
561         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasTypedEntitiesSemantics> some
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
562
563
564 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithAbstracClass>
565
566     EquivalentTo:
567         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
568          and (<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasInheritancePropagationSemantics>
              some <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#AbstracClass>)
569
570
571 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithInheritance>
572
573     SubClassOf:
574         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
575
```

FIGURE 46 – Partie 11

```
575
576
577 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithMultipleInheritance>
578
579     SubClassOf:
580         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithInheritance>
581
582     DisjointWith:
583         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithSimpleInheritance>
584
585
586 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithSimpleInheritance>
587
588     SubClassOf:
589         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithInheritance>
590
591     DisjointWith:
592         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LanguageWithMultipleInheritance>
593
594
595 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#LateBinding>
596
597     SubClassOf:
598         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>
599
600     DisjointWith:
601         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#EarlyBinding>
602
603
604 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Method>
605
606     SubClassOf:
607         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
608
609
610 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#MethodLock>
611
612     SubClassOf:
613         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#FullLock>
614
615     DisjointWith:
616         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#ClassLock>
617
618
619 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
620
621     SubClassOf:
622         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>
623
624     DisjointWith:
625         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
626
```

FIGURE 47 – Partie 12

```
626
627
628     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
629
630         SubClassOf:
631             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Element>
632
633         DisjointWith:
634             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
635
636
637     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableEvent>
638
639         SubClassOf:
640             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
641
642
643     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableIndexer>
644
645         SubClassOf:
646             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
647
648
649     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableMethod>
650
651         SubClassOf:
652             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
653
654
655     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableProperty>
656
657         SubClassOf:
658             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
659
660
661     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableVariable>
662
663         SubClassOf:
664             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableElement>
665
666
667     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PartialLock>
668
669         SubClassOf:
670             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PreventFurtherInheritance>
671
672         DisjointWith:
673             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#FullLock>
674
675
676     Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PreventFurtherInheritance>
677
678         SubClassOf:
679             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#InheritancePropagation>
680
681         DisjointWith:
682             <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#ForceFurtherInheritance>
683
```

FIGURE 48 – Partie 13

```
683
684
685 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#PrimitiveType>
686
687     SubClassOf:
688         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
689
690
691 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Property>
692
693     SubClassOf:
694         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
695
696
697 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#StaticElement>
698
699     SubClassOf:
700         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
701
702
703 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#StaticallyTypedLanguage>
704
705     SubClassOf:
706         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>
707
708     DisjointWith:
709         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#DynamicallyTypedLanguage>
710
711
712 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Type>
713
714     SubClassOf:
715         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#isTypeOf> exactly 1
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
716
717
718 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#TypedEntity>
719
720     SubClassOf:
721         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#hasType> exactly 1
            <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Type>
722
```

FIGURE 49 – Partie 14

```
722
723
724 Class: <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Variable>
725
726     SubClassOf:
727         <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#NonOverridableElement>
728
729
730 DisjointClasses:
731
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Binding>,<http://www.semanticweb.org/utilisateur
        /ontologies/2023/0/LanguageOntology#Element>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Inh
        eritancePropagation>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Language>,<http://www.seman
        ticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Type>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/L
        anguageOntology#TypedEntity>
732
733 DisjointClasses:
734
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Class>,<http://www.semanticweb.org/utilisateur/o
        ntologies/2023/0/LanguageOntology#Interface>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Pri
        mitiveType>
735
736 DisjointClasses:
737
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Constant>,<http://www.semanticweb.org/utilisateu
        r/ontologies/2023/0/LanguageOntology#Event>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Inde
        xer>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Method>,<http://www.semanticweb.org/utilisa
        teur/ontologies/2023/0/LanguageOntology#Property>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntolog
        y#StaticElement>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Variable>
738
739 DisjointClasses:
740
        <http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#Label>,<http://www.semanticweb.org/utilisateur/o
        ntologies/2023/0/LanguageOntology#OverridableEvent>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntol
        ogy#OverridableIndexer>,<http://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableMethod>,<http
        ://www.semanticweb.org/utilisateur/ontologies/2023/0/LanguageOntology#OverridableProperty>,<http://www.semanticweb.org/util
        isateur/ontologies/2023/0/LanguageOntology#OverridableVariable>
741
```

FIGURE 50 – Partie 15