HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

RAPPORT DE PROJET

Moteur de validation des contraintes qualitatives

Loïc HERMAN

Rapport final sur l'implémentation du langage

StandardQL

pour la réalisation du

Moteur de validation des contraintes de qualité du Registre fédéral des Bâtiments et des Logements (RegBL)

dans le cadre du

Développement de l'implémentation cantonale du registre dans le système d'information des bâtiments (SIBAT)

pour la

Direction du Cadastre et de la Géoinformation (DCG)

12 juin 2025

HEIG VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Table des matières

1 Étapes de configuration et validation des contrôles de qualité sur les bâtiments suite aux modifications par les communes

Le rapport intermédiaire du projet a établi les fondements théoriques de StandardQL autour de trois axiomes principaux : l'approche déclarative pour l'expression des contraintes, la conceptualisation des identifiants comme fonctions appliquées aux entités contextuelles, et l'extensibilité du langage via son contexte d'exécution. Ces principes, inspirés de la programmation logique sous contraintes, visaient à créer un pont entre l'expression informelle des règles métier et leur implémentation technique.

Dans ce chapitre, nous allons présenter la mise en œuvre concrète de ces concepts théoriques dans le contexte opérationnel du système SIBAT. Nous détaillerons brièvement le processus depuis la saisie des mutations par les communes vaudoises jusqu'à la validation finale des données avant leur transmission au RegBL fédéral. L'accent sera mis sur l'architecture du système de validation, illustrant comment StandardQL s'intègre naturellement dans la chaîne de traitement des données tout en restant transparent pour les utilisateurs finaux. Nous analyserons également les mécanismes de configuration des règles de qualité, démontrant la flexibilité du langage face aux exigences évolutives du domaine métier.

1.1 Vue d'ensemble du processus de validation

La figure 1.1 illustre l'architecture générale du système de validation des données bâtimentaires dans SIBAT. Le processus s'articule autour de quatre étapes principales orchestrées par le moteur de validation.

1. **Saisie et transmission des données** : Les communes vaudoises accèdent au portail des prestations pour saisir les mutations concernant les bâtiments de leur territoire. Ces modifications, qu'elles concernent des nouvelles constructions, des transformations ou des démolitions, sont transmises au moteur de validation.
2. **Configuration préalable du système** : En amont, le gestionnaire applicatif configure le moteur avec l'ensemble des règles de qualité. Cette configuration comprend les ~ 500

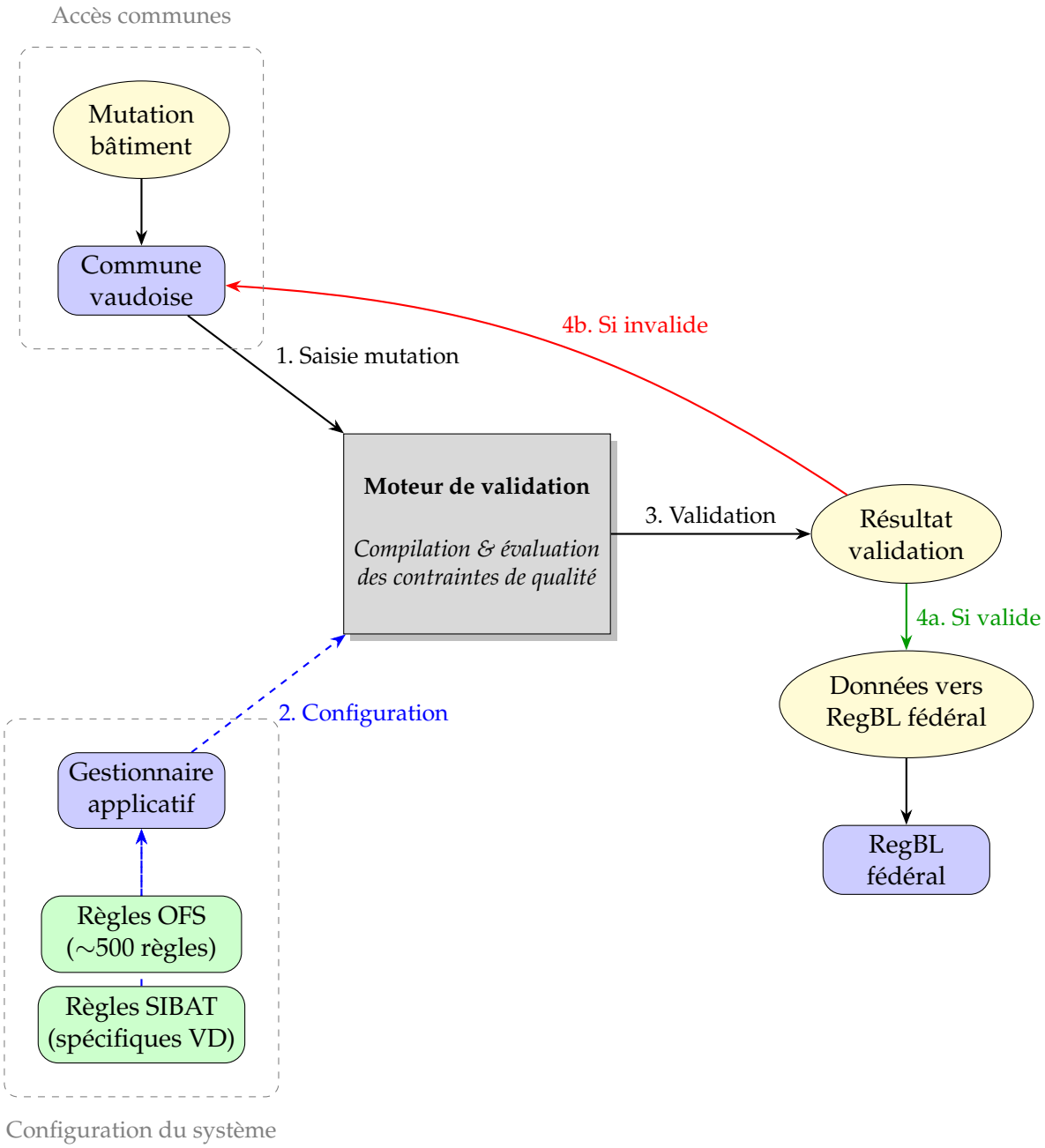

FIGURE 1.1 – Exécution standard d'une mutation des données d'un bâtiment dans SIBAT pour envoi au RegBL

règles définies par l'OFS ainsi que les règles spécifiques au canton de Vaud développées pour SIBAT. Ces règles, exprimées en StandardQL, couvrent les trois catégories de validation identifiées : spécifications techniques, obligations d'annonce et exigences de qualité.

3. **Validation par le moteur StandardQL** : Le cœur du système compile dynamiquement les règles configurées et les applique aux données soumises. Cette étape, transparente pour l'utilisateur, mobilise l'ensemble des concepts théoriques développés : évaluation déclarative des contraintes, résolution des identifiants comme fonctions contextuelles, et extension du langage via les fonctions métier.
4. **Traitement des résultats** : Selon l'issue de la validation, deux chemins sont possibles. En cas de conformité, les données seront finalement transmises au RegBL fédéral. En cas de non-conformité, un rapport détaillé des erreurs est retourné à la commune pour correction.

Cette architecture positionne le moteur et son langage, StandardQL, comme l'élément central du système, garantissant la cohérence et la fiabilité des données tout en préservant la simplicité d'utilisation pour les communes.

1.2 Architecture du moteur de validation

Le moteur de validation constitue l'implémentation concrète des concepts théoriques de StandardQL. L'architecture adoptée, décrite en figure 1.2, suit un modèle de compilation en deux phases : compilation par le module Haskell vers une représentation intermédiaire, puis une évaluation avec Java sur les données métier.

Le parser StandardQL, développé avec la bibliothèque Megaparsec, transforme les expressions déclaratives en arbre syntaxique abstrait (AST). Cette première phase effectue l'élaboration syntaxique, traduisant notamment les constructions *Condition* en expressions binaires équivalentes. Le compilateur produit une représentation JSON sérialisée, assurant l'interopérabilité avec l'environnement Java.

Un module Java désérialise ensuite l'AST et l'évalue via le pattern visiteur. Cette architecture permet l'extension du langage par injection de dépendances : les fonctions du langage (`COUNT`, `SUM`, ...) sont automatiquement découvertes et intégrées au contexte d'évaluation.

Un service dédié implémente le concept d'identifiants comme fonctions. Utilisant la réflexion Java sur les records de modèle, il résout dynamiquement les attributs selon leur contexte d'entité. Les relations inter-entités sont gérées par des composants spécialisés.

Le système exploite les threads virtuels pour l'évaluation en parallèle des règles. Chaque règle compilée devient un bean Spring singleton, permettant son exécution concurrente sur les données.

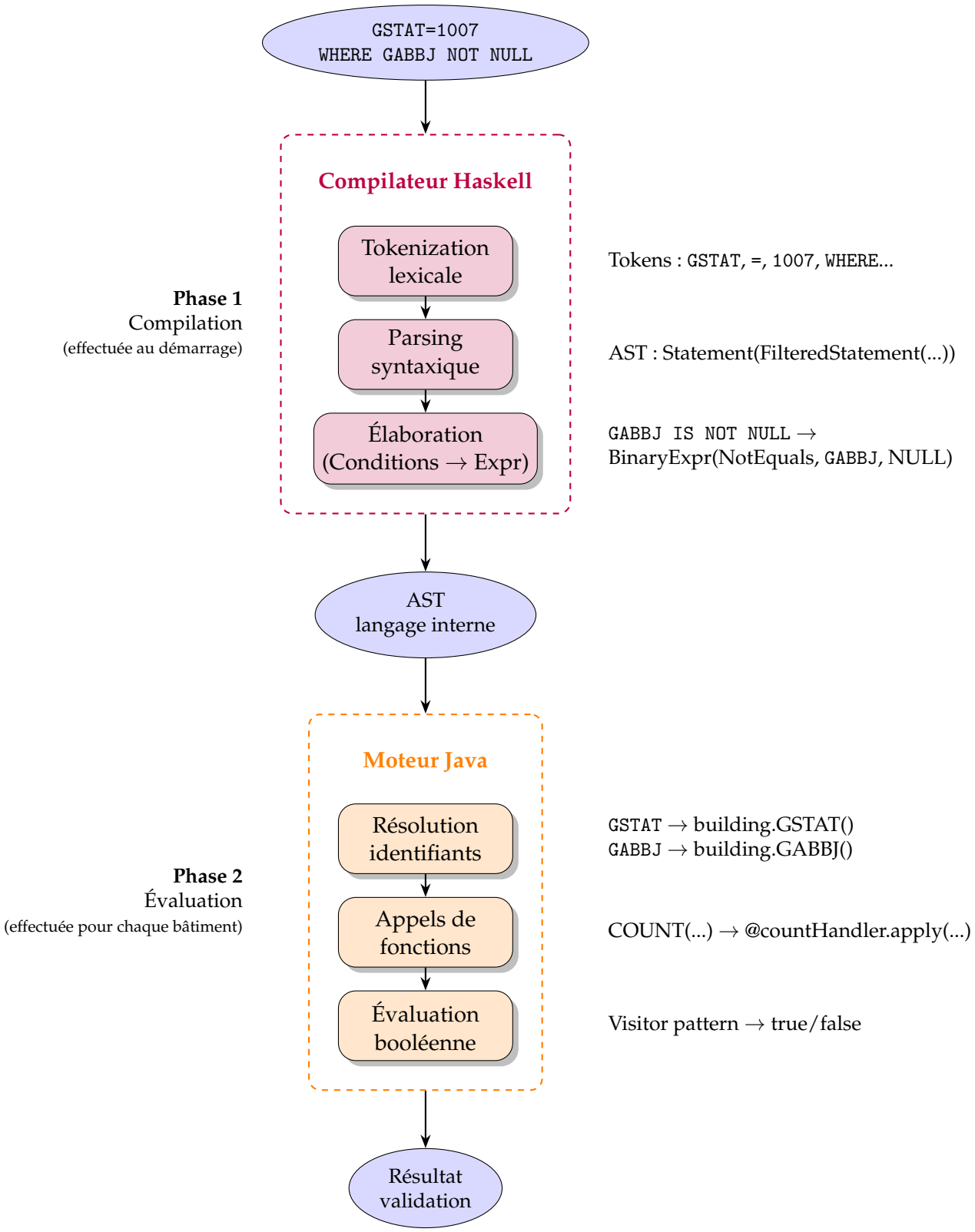

FIGURE 1.2 – Processus simplifié de la compilation d'une expression StandardQL telle que donnée dans la configuration vers une évaluation concrète d'un objet

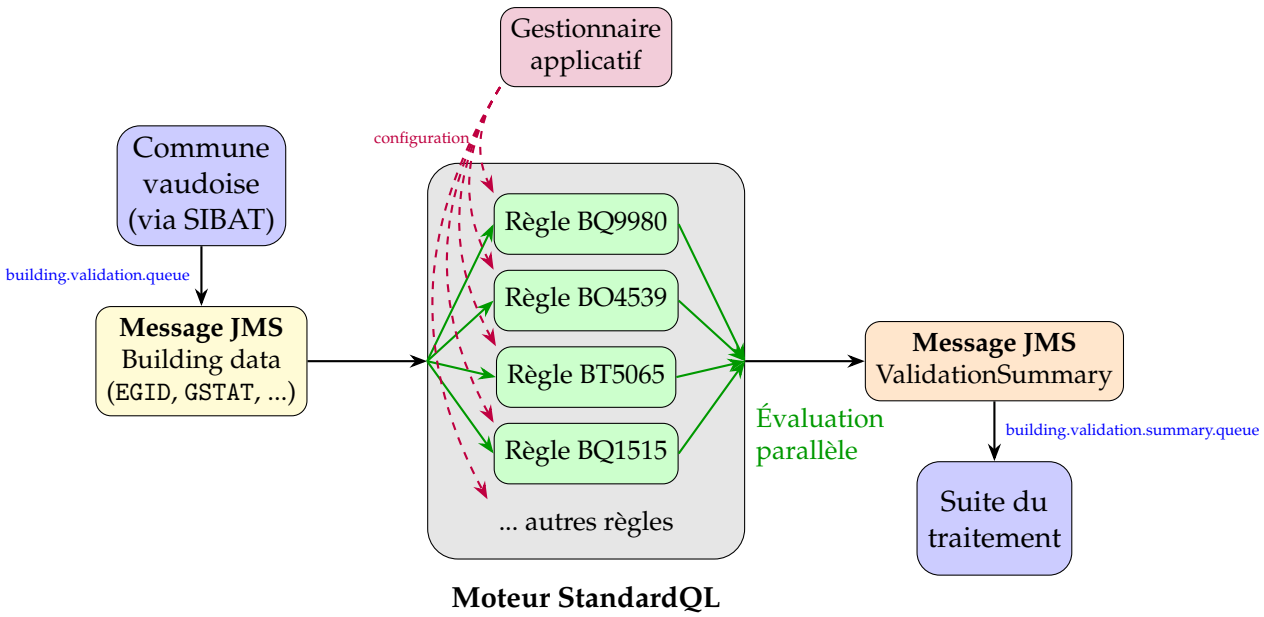

FIGURE 1.3 – Visualisation de la maquette inspirée du système SIBAT réel pour faire interface au moteur de validation

1.3 Traitement distribué et parallèle des règles de validation

La figure 1.3 illustre l'architecture de traitement des mutations bâtimentaires par le moteur StandardQL. Le système repose sur une approche distribuée exploitant ActiveMQ pour le découplage et la parallélisation. Comme décrit dans le rapport intermédiaire, ceci est une version simplifiée du processus réel pour SIBAT. L'objectif ici est d'avoir une interface similaire qui fasse objet de maquette pour évaluer comment fonctionnerait le langage StandardQL dans un tel écosystème.

Lorsqu'une commune soumet une mutation, les données du bâtiment sont sérialisées dans un message JMS et envoyées vers la queue `building.validation.queue`. À la réception du message, le service va charger dynamiquement toutes les règles applicables via le registre des règles. Chaque règle, préalablement compilée en StandardQL et instanciée comme composant Spring, est exécutée en parallèle via les threads virtuels de Java 21. Cette parallélisation exploite pleinement les ressources système puisque l'évaluation des règles est généralement indépendante.

Le moteur illustre la position centrale du langage dans l'architecture. Toutes les règles, qu'elles proviennent de l'OFS ou soient spécifiques à SIBAT, sont exprimées dans le même formalisme StandardQL. Cette uniformisation permet au gestionnaire applicatif de configurer, modifier ou ajouter des règles sans intervention technique, le moteur gérant automatiquement leur compilation et leur intégration.

Les résultats individuels sont collectés et agrégés dans un résumé contenant les statistiques globales (règles passées, échouées, erreurs techniques). Ce résumé est retourné via la queue `building.validation.summary.queue`, bouclant le cycle de validation.

2 Implémentation d'un compilateur pour le langage StandardQL en Haskell

L'implémentation du compilateur StandardQL constitue le point central de notre système de validation. Ce chapitre détaillera l'architecture et les choix techniques qui ont guidé le développement du compilateur, depuis l'analyse syntaxique jusqu'à la production d'une représentation intermédiaire exploitable par le moteur d'exécution Java.

2.1 Architecture générale du compilateur

Le compilateur StandardQL a été conçu selon une architecture modulaire classique, comprenant trois phases principales : l'analyse lexicale et syntaxique, l'élaboration, et la sérialisation. Cette approche permet une séparation claire des responsabilités et facilite la maintenance et l'évolution du système.

Le choix du langage Haskell pour l'implémentation du compilateur s'est imposé naturellement pour plusieurs raisons. D'abord, la nature fonctionnelle du langage s'aligne parfaitement avec le paradigme déclaratif de StandardQL. Ensuite, l'écosystème Haskell offre des bibliothèques matures pour la construction de parsers, notamment Megaparsec, qui permet d'implémenter des analyseurs syntaxiques robustes et expressifs.

2.2 Analyse syntaxique par combinateurs monadiques

L'analyse syntaxique constitue la première étape du processus de compilation. Nous avons opté pour une approche basée sur les combinateurs de parsers monadiques, en utilisant la bibliothèque Megaparsec.

Les combinateurs de parsers permettent de construire l'analyseur syntaxique de manière compositionnelle, où chaque règle de la grammaire formelle (définie en annexe A) est implémentée comme une fonction de parsing distincte. Ces fonctions peuvent ensuite être combinées pour former des parsers plus complexes, reflétant directement la structure hiérarchique de la grammaire.

L'utilisation de la monade `Parser` de Megaparsec offre une gestion élégante des erreurs et du backtracking, permettant de produire des messages d'erreur précis et contextuels.

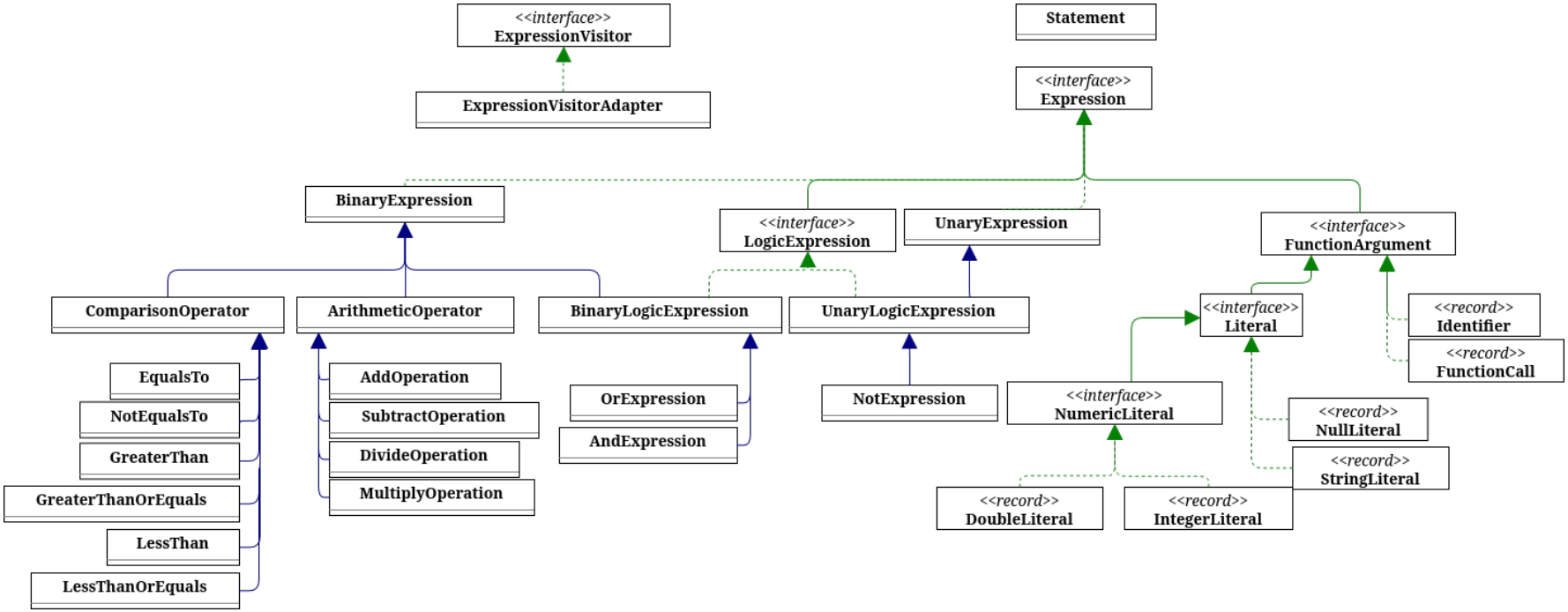

FIGURE 2.1 – Diagramme de classes de l'arbre syntaxique abstrait représentant le langage interne du StandardQL

2.3 Représentation intermédiaire et élaboration

Une particularité importante de notre compilateur est la transformation immédiate des tokens analysés en une représentation intermédiaire, sous forme d'arbre syntaxique abstrait (AST). La structure de cet AST, dont sa contrepartie Java est illustrée dans la figure 2.1, a été conçue pour être à la fois expressive et facilement sérialisable.

Le processus d'élaboration intervient directement pendant l'analyse syntaxique. Cette approche diffère des compilateurs traditionnels qui maintiennent généralement une séparation stricte entre les phases d'analyse et de transformation. Dans notre cas, certaines constructions syntaxiques, notamment les conditions, sont immédiatement transformées en leur forme canonique.

2.3.1 Élaboration des conditions

Les conditions dans StandardQL représentent un sucre syntaxique permettant d'exprimer de manière concise des contraintes sur un attribut unique. Par exemple, l'expression `GSTAT IN (1000-2000)` est plus lisible que son équivalent développé `GSTAT >= 1000 AND GSTAT <= 2000`.

L'élaboration des conditions s'effectue selon les règles suivantes :

— `NULL` → génération d'une expression de comparaison avec la valeur nulle

— `BETWEEN x AND y` → conjonction de deux comparaisons

— `IN (v1-v2|...|vn)` → disjonction de comparaisons d'égalité

— `MATCHES expr` → application de l'expression avec fermeture lexicale sur l'identifiant

Cette transformation immédiate présente plusieurs avantages. Elle simplifie considérablement les phases ultérieures du compilateur en réduisant le nombre de cas à traiter, et elle garantit que la sémantique des conditions est correctement préservée dès le début du processus de compilation.

2.4 Interface en ligne de commande

Pour permettre l'intégration du compilateur Haskell dans l'écosystème Java, nous avons développé une interface en ligne de commande (CLI) simple mais efficace. Cette interface accepte une expression StandardQL en entrée et produit une représentation JSON de l'AST élaboré en sortie.

Le format JSON a été choisi pour sa simplicité et son support natif dans la plupart des langages de programmation. La structure de sérialisation a été conçue pour être à la fois compacte et auto-descriptive, facilitant la désérialisation côté Java sans nécessiter de schéma externe complexe.

3 Définition extensible de l'exécution dynamique des règles de qualité en Java

L'implémentation du moteur de validation constitue la partie opérationnelle du système StandardQL. Cette architecture modulaire a été conçue pour offrir une séparation claire des responsabilités tout en permettant une extensibilité maximale du système d'évaluation.

3.1 Architecture modulaire du moteur

Le moteur de validation est structuré en trois modules distincts, chacun ayant une responsabilité bien définie.

3.1.1 Module de modélisation (model)

Ce module constitue le socle commun partagé entre tous les composants du système. Il définit les structures de données représentant les entités du RegBL (bâtiments, logements, entrées, etc.) ainsi que les interfaces nécessaires à l'exécution des règles. Cette approche centralisée garantit la cohérence des données à travers l'ensemble de l'application.

3.1.2 Module de compilation (compiler)

Le module de compilation agit comme une passerelle entre le compilateur Haskell et le moteur Java. Il gère deux responsabilités principales :

— La désérialisation du langage interne produit par le parser Haskell en classes Java immutables

— L'implémentation du patron visiteur permettant le parcours et l'évaluation de l'arbre syntaxique

L'utilisation d'objets immutables garantit la sûreté en environnement concurrent, aspect crucial pour l'évaluation parallèle des règles.

3.1.3 Module moteur (engine)

Le cœur du système est une application Spring Boot complète, conçue pour être déployée comme un microservice autonome. Il écoute une file de messages ActiveMQ et orchestre la validation asynchrone des règles sur les objets reçus.

Il est important de noter que cette implémentation constitue une version simplifiée du moteur réel utilisé dans le système SIBAT du canton de Vaud. Les mécanismes de communication et de gestion des files ont été volontairement épurés pour se concentrer sur l'essence du paradigme et de l'évaluation du langage. L'interface maquettisée utilisée dans notre implémentation a été développée à neuf et se distingue complètement des systèmes déjà mis en place pour l'État de Vaud.

3.2 Composants de l'évaluation dynamique

L'évaluation des règles StandardQL repose sur quatre composants essentiels qui collaborent pour transformer une expression déclarative en validation concrète.

3.2.1 Service de résolution des identifiants

Le `IdentifierResolutionService` maintient un registre associant chaque identifiant StandardQL à son attribut correspondant dans les objets Java. Cette approche par réflexion permet d'accéder dynamiquement aux valeurs des attributs sans code spécifique par règle. Le service utilise les records de Java pour extraire les valeurs de manière type-safe depuis les instances données dans le contexte.

3.2.2 Résolveurs de relations

Les `RelationObjectResolver` gèrent la navigation entre les entités liées. Chaque résolveur définit explicitement une relation source-cible (par exemple, bâtiment vers logements) et implémente la logique d'extraction des objets liés. L'interface à implémenter n'est pas très complexe, deux méthodes permettent de récupérer les types d'entité source et cible, et une méthode d'application reçoit le contexte d'évaluation pour récupérer l'entité de ce qui a été préalablement ajouté audit contexte.

Ces instances de résolution peuvent être créées séparément et chaque instance est responsable de traiter une relation. Elles sont ensuite injectées dans les services vus ci-après lorsque c'est nécessaire selon les informations d'une expression.

3.2.3 Gestionnaires de fonctions

La grammaire StandardQL ne définit pas de fonctions à proprement parlé, mais elle ajoute les moyens de faire les appels. La responsabilité de la création des fonctions dans le contexte d'exécution revient donc au consommateur du langage. En l'occurrence, les fonctions StandardQL sont implémentées comme des beans Spring annotés `@FunctionHandler`. Chaque fonction expose deux méthodes essentielles :

— `validate()` : vérifie que les arguments fournis sont du type attendu

— `execute()` : applique la transformation ou l'agrégation sur les données

Cette architecture permet l'ajout de nouvelles fonctions par simple ajout d'un bean, illustrant parfaitement l'extensibilité du système.

3.2.4 Service d'évaluation

Finalement, le `RuntimeEvaluationService` orchestre l'évaluation complète d'une règle. Implémenté comme un visiteur de l'arbre syntaxique, il :

1. Parcourt récursivement l'expression StandardQL
2. Résout les identifiants vers leurs valeurs concrètes
3. Applique les fonctions et opérateurs
4. Retourne un résultat booléen indiquant la validité

La gestion des erreurs est intégrée à chaque niveau, transformant les exceptions techniques en statuts de validation appropriés.

3.2.5 Initialisation des règles

Au démarrage de l'application, un service d'initialisation charge toutes les règles définies dans la configuration YAML. Pour chaque règle :

1. L'expression StandardQL est compilée via le service de compilation
2. Une instance `CompiledRule` est créée, encapsulant l'AST compilé et les métadonnées
3. Le bean est enregistré dans le contexte Spring sous un identifiant unique

Cette approche garantit que toute erreur de syntaxe est détectée au démarrage plutôt qu'à l'exécution, et permet de faire une seule passe de compilation pour toutes les règles définies. Un autre avantage est que le service étant derrière une queue ActiveMQ, lors de la mise à jour des règles, les traitements seront mis en pause jusqu'à ce que toutes les règles aient pu être rafraîchies.

4 Conclusion

4.1 Retour d'expérience

Le développement de StandardQL et de son moteur d'évaluation a confirmé la pertinence d'une approche déclarative pour l'expression des contraintes de qualité. L'architecture mise en place démontre qu'il est possible de concilier simplicité d'expression et puissance d'évaluation tout en maintenant des performances acceptables.

Exemples concrets d'utilisation

L'expressivité du langage se révèle particulièrement dans la gestion des relations entre entités. Nous discuterons ici de quelques exemples tirés de l'implémentation.

Dans le listing 1, la règle vérifie qu'un bâtiment « d'habitations provisoire » contienne effectivement une seule entrée. La fonction `COUNT` s'applique automatiquement sur la collection des identifiants fédéraux des entrées liés au bâtiment, illustrant l'élégance de l'approche fonctionnelle.

La règle du listing 2 est plus complexe, cette règle permet de valider que la somme des surfaces des logements ne dépasse pas la surface au sol multipliée par le nombre d'étages, une contrainte de cohérence physique importante.

L'utilisation de fonctions d'agrégation sur les entités liées (comme `MIN(WBAUJ)` pour trouver l'année de construction du plus ancien logement) permet d'exprimer des contraintes inter-entités qui auraient nécessité des boucles explicites dans une approche impérative.

Listing 1 Code StandardQL de la règle BQ1515

```
COUNT(EDID)=1
WHERE GKAT=1010
```

Listing 2 Code StandardQL de la règle VDBQ1785

```
SUM(WAREA) <= GAREA * GASTW
WHERE GKAT IN (1020|1030)
```

Listing 3 Exemple de fonction d'ordre supérieur

```
COUNT(FILTER(WSTAT, \x. x = 3004)) > 0
WHERE GSTAT = 1004
-- Alternative par expression
COUNT(WSTAT WHERE WSTAT = 3004) > 0
WHERE GSTAT = 1004
```

4.2 Améliorations possibles

Fonctions d'ordre supérieur

L'évolution naturelle du langage serait l'introduction de fonctions partielles et de lambdas comme paramètres de fonctions. Cette extension permettrait des constructions plus sophistiquées comme celle présentée dans le listing 3.

Cette syntaxe hypothétique permettrait de compter uniquement les logements existants dans un bâtiment existant, offrant un contrôle plus fin sur les agrégations.

Optimisations de performance

L'évaluation actuelle, bien que fonctionnelle, pourrait bénéficier d'optimisations :

— Mise en cache des résolutions d'identifiants fréquemment utilisés
— Évaluation paresseuse des expressions avec court-circuit plus agressif
— Compilation vers du bytecode Java pour les règles critiques

4.3 Bilan final

StandardQL démontre qu'il est possible de créer un langage dédié qui soit à la fois accessible aux experts métier et suffisamment expressif pour couvrir la complexité des contraintes du RegBL. L'architecture modulaire du moteur d'évaluation, couplée à l'utilisation judicieuse des capacités modernes de Java (records, threads virtuels, pattern matching), produit un système robuste et maintenable.

La séparation claire entre la logique métier (exprimée en StandardQL) et l'infrastructure technique (le moteur Java) constitue un atout majeur pour l'évolution future du système. Cette approche permet d'envisager sereinement l'ajout de nouvelles contraintes, l'extension vers d'autres registres, ou même l'adaptation à des contextes réglementaires différents.

Le succès de cette implémentation réside finalement dans sa capacité à transformer un problème technique complexe en un outil accessible, permettant aux experts du domaine de reprendre le contrôle sur la définition et l'évolution des règles de qualité qui régissent leurs données.

A Grammaire et typage de StandardQL

Cette annexe provient du rapport intermédiaire concernant la théorie du langage StandardQL, dans lequel nous avons défini la grammaire et le typage attendus d'une expression valide.

A.1 Grammaire formelle du langage

La grammaire de StandardQL est conçue pour être rigoureuse mais intuitive, permettant autant une analyse syntaxique efficace qu'une utilisation aisée par des non-spécialistes. Nous présentons ci-dessous la grammaire formelle du langage, exprimée en notation EBNF.

A.1.1 Structure générale

Un programme StandardQL se compose d'une unique déclaration (Statement), qui représente une règle de qualité, suivie de la fin du fichier. Cette déclaration comprend une expression principale et, optionnellement, une expression de filtrage introduite par le mot-clé `WHERE`. La structure est donnée en figure A.1.

A.1.2 Expressions

Les expressions dans StandardQL décrites en figure A.2 suivent une hiérarchie classique avec des priorités d'opérateurs. Les opérateurs logiques (`AND`, `OR`) et relationnels (`=`, `!=`, $<$, $>$, etc.) permettent de combiner des expressions plus simples. Le langage introduit également le concept de ConditionExpr, qui permet d'exprimer des contraintes sur un attribut spécifique de manière concise. La table en figure A.3 décrit les termes primaires du langage, ce sont les termes pour lesquels il n'est pas nécessaire de spécifier la préséance relative d'autres termes primaires, car la syntaxe ne sera jamais ambiguë.

Program	$\mathscr{P}$	::=	$\mathcal{S}$ `EOF`	constraint
Statement	$\mathcal{S}$	::=	e	simple rule
		|	e `WHERE` e	filtered rule

FIGURE A.1 – Structure du langage externe de StandardQL

Expression	e	::=	e_L	simple term
		|	e `OR` e_L	or expression
LogicalTerm	e_L	::=	e_R	simple term
		|	e_L `AND` e_R	and expression
RelationalExpr	e_R	::=	e_C	simple term
		|	e_C `=` e_C	equality
		|	e_C (`!=` | `<>`) e_C	inequality
		|	e_C (`<` | `<=` | `>` | `>=`) e_C	comparison
		|	e_C (`+` | `-`) e_C	arithmetic I
		|	e_C (`*` | `/`) e_C	arithmetic II
ConditionExpr	e_C	::=	$\mathcal{P}$	primary term
		|	$\mathcal{I}$ $\wedge_c$	condition on term

FIGURE A.2 – Expressions du langage externe de StandardQL

Primary	$\mathcal{P}$	::=	$\supset_{\mathrm{E}}$	function call
		|	v	valid value
		|	(e)	inner expression
FunctionCall	$\supset_{\mathrm{E}}$	::=	$\mathcal{I}$ ([ArgumentList])	function call
ArgumentList		::=	($\mathcal{I}$ | v | $\supset_{\mathrm{E}}$)	single argument
		|	($\mathcal{I}$ | v | $\supset_{\mathrm{E}}$) , ArgumentList	multiple arguments

FIGURE A.3 – Termes primaires du langage externe de StandardQL

ConditionList	$\wedge_c$	::=	c	simple condition
		|	c `AND` $\wedge_c$	and condition
		|	c `OR` $\wedge_c$	or condition
Condition	c	::=	[`IS`] `NULL`	nullity
		|	[`IS`] `NOT NULL`	non-nullity
		|	`=` v	equality
		|	(`!=` | `<>`) v	inequality
		|	(`<` | `<=` | `>` | `>=`) v	comparison
		|	`BETWEEN` v `AND` v	simple domain
		|	`IN` ($\cup_v$)	value domain
		|	`NOT IN` ($\cup_v$)	inverse value domain
		|	[`MATCHES`] (e)	closure-bound inner expression
ValueList	$\cup_v$	::=	v [`-` v]	single range
		|	v [`-` v] `'|'` $\cup_v$	multiple ranges

FIGURE A.4 – Conditions du langage externe de StandardQL

$$
\begin{array}{lcll}
\text{Value} & v & ::= & \text{NumericLiteral} \\
 & & \mid & \text{StringLiteral} \\
\text{Identifier} & \mathcal{I} & ::= & \text{Letter } \{ \text{ Letter } \mid \text{ Digit } \mid \text{ \_ } \}
\end{array}
$$

FIGURE A.5 – Identifiants et littéraux du langage externe de StandardQL

$$
\begin{array}{rcl}
\tau & ::= & \textbf{int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{bool} \mid \textbf{date} \\
 & \mid & [\tau] \mid \textbf{entity}(\texttt{e}) \mid \textbf{null}
\end{array}
$$

FIGURE A.6 – Types utilisés dans le typage dynamique de StandardQL

A.1.3 Conditions

Les conditions représentent des contraintes spécifiques appliquées à un identifiant (attribut). Elles peuvent exprimer des tests de nullité, des comparaisons avec des valeurs, ou des tests d'appartenance à un ensemble ou à un intervalle. Cela constitue un sucre syntaxique par rapport aux expressions de base. Leur rôle est de simplifier l'écriture des contraintes qui s'appliquent à un même attribut, réduisant ainsi la duplication de code. La table en figure A.4 en donne un exemple prévisionnel.

Ce mécanisme peut être étendu avec des syntaxes supplémentaires tant qu'elles peuvent être réduites à des expressions simples sur l'identifiant concerné. Par exemple, la condition `IN <value-range>` se traduit en arrière-plan par une série d'expressions de comparaison avec des opérateurs logiques.

A.1.4 Identifiants et littéraux

La figure A.5 définit pour finir la syntaxe des identifiants (noms d'attributs et de fonctions) et des littéraux (valeurs numériques et chaînes de caractères). On ne répète pas dans la syntaxe la notion triviale de nombres et de strings (les littéraux sous forme de strings sont entourés de guillemets et les nombres sont représentés tels quels).

A.2 Système de types

Le système de types de StandardQL est conçu pour être simple mais expressif, permettant la manipulation des différents types de données présents dans le RegBL. La figure A.6 définit les types premiers utilisés par le langage.

Cette section décrira le système prévisionnel de types qui devra être mis en place pour répondre aux exigences placées ci-avant.

A.2.1 Identifiants comme fonctions

Le système de types de StandardQL considère chaque identifiant comme une fonction qui s'applique à l'entité contextuelle courante. Nous formalisons cette notion à travers un système de jugement de types où le contexte d'évaluation joue un rôle central.

$$\frac{\Gamma(\text{id}) = \textbf{entity}(e) \to \tau}{\Gamma \vdash \text{id} : \textbf{entity}(e) \to \tau} \quad \text{(ID)} \tag{A.1}$$

Dans un contexte d'évaluation Γ, un identifiant id est typé comme une fonction qui prend une entité de type e et retourne une valeur de type τ. Le type τ et l'entité e sont déterminés dynamiquement au runtime avec le contexte d'évaluation.

Par exemple, si nous traitons une règle dont l'attribut principal référence un bâtiment. Le jugement suivant s'applique :

$$\Gamma \vdash \texttt{GSTAT} : \textbf{entity}(\texttt{BUILDING}) \to \textbf{int} \tag{A.2}$$

Sous forme textuelle, on dira que l'attribut représentant le statut du bâtiment (`GSTAT`) est disponible dans le contexte Γ et sa fonction associée retourne le statut sous forme d'entier lorsqu'elle est appelée avec le bâtiment en cours de traitement.

Pour les identifiants qui référencent des attributs d'entités liées (notées e_k), nous introduisons une règle d'inférence supplémentaire qui modélise la navigation relationnelle :

$$\frac{\Gamma(\text{id}) = \textbf{entity}(e) \to \textbf{entity}(e_k) \to \tau}{\Gamma \vdash \text{id} : \textbf{entity}(e) \to [\tau]} \quad \text{(REL-ID)} \tag{A.3}$$

Cette règle capture le fait que si un identifiant id est lié dans le contexte Γ à une fonction d'ordre supérieur déclarant que l'attribut mentionné fait partie d'une relation sur une entité liée e_k de l'entité principale e, le programme à l'exécution retournera une liste $[\tau]$ de valeurs correspondant à l'attribut donné. La norme de la liste dépend de la cardinalité de la relation entre les deux entités.

Dans le même exemple qu'avant, le jugement suivant s'applique :

$$\frac{\Gamma(\texttt{WSTAT}) = \textbf{entity}(\texttt{BUILDING}) \to \textbf{entity}(\texttt{DWELLING}) \to \textbf{int}}{\Gamma \vdash \texttt{WSTAT} : \textbf{entity}(\texttt{BUILDING}) \to [\textbf{int}]} \tag{A.4}$$

Cette liaison indique que `WSTAT` est une fonction qui, appliquée à un bâtiment, retourne une fonction qui, appliquée à un logement, retourne un nombre (le statut du logement). Dans le contexte d'une règle qui s'applique aux bâtiments, l'appel à `WSTAT` retourne donc une liste de valeurs numériques représentant les statuts de tous les logements associés au bâtiment.

A.2.2 Fonctions d'agrégation et autres fonctions utilitaires

Les fonctions d'agrégation et utilitaires sont essentielles pour manipuler les données dans StandardQL. Nous formalisons leurs types à travers des règles d'inférence qui spécifient les types d'entrée et de sortie.

$$\frac{\Gamma \vdash f : \tau_1 \times \tau_2 \times \ldots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash f(e_1, e_2, \ldots, e_n) : \tau} \text{ (APP)} \tag{A.5}$$

$$\frac{\Gamma \vdash v : \textbf{string}}{\Gamma \vdash \text{LENGTH}(v) : \textbf{int}} \text{ (LENGTH)} \tag{A.6}$$

$$\frac{\Gamma \vdash v : [\tau]}{\Gamma \vdash \text{COUNT}(v) : \textbf{int}} \text{ (COUNT)} \tag{A.7}$$

$$\frac{\Gamma \vdash v : [\textbf{int}]}{\Gamma \vdash \text{SUM}(v) : \textbf{int}} \text{ (SUM)} \qquad \frac{\Gamma \vdash v : [\textbf{float}]}{\Gamma \vdash \text{SUM}(v) : \textbf{float}} \text{ (SUMF)} \tag{A.8}$$

$$\frac{\Gamma \vdash v : [\textbf{int}]}{\Gamma \vdash \text{MAX}(v) : \textbf{int}} \text{ (MAX)} \qquad \frac{\Gamma \vdash v : [\textbf{float}]}{\Gamma \vdash \text{MAX}(v) : \textbf{float}} \text{ (MAXF)} \tag{A.9}$$

$$\frac{\Gamma \vdash v : [\textbf{int}]}{\Gamma \vdash \text{MIN}(v) : \textbf{int}} \text{ (MIN)} \qquad \frac{\Gamma \vdash v : [\textbf{float}]}{\Gamma \vdash \text{MIN}(v) : \textbf{float}} \text{ (MINF)} \tag{A.10}$$

Les jugements A.7 à A.10 permettent les opérations d'agrégation sur les listes. Le jugement A.6 définit la fonction sur les chaînes de caractères. Enfin, le jugement A.5 permet la définition d'autres fonctions génériques comme le produit de types vers un type résultant.

Dans le cas particulier des conditions `MATCHES`, une fermeture lexicale est créée pour les appels de fonction qui apparaissent dans l'expression de correspondance. Lorsqu'un identifiant est utilisé avec une condition `MATCHES`, cet identifiant (qui est lui-même une fonction appliquée à l'entité propriétaire) est implicitement fourni comme premier paramètre à tout appel de fonction lié dans l'expression de correspondance qui n'a pas d'argument explicite. Cette règle peut être formalisée comme suit :

$$\frac{\begin{array}{l}\Gamma \vdash \text{id} : \textbf{entity}(e) \to \tau_1 \\ \Gamma \vdash f : \tau_1 \to \tau_2 \\ \Gamma \vdash \text{expr}[f() \mapsto f(id)] : \textbf{bool}\end{array}}{\Gamma \vdash \text{id MATCHES expr} : \textbf{bool}} \text{ (MATCHES)} \tag{A.11}$$

Par exemple, dans une expression comme `LPARZ NULL OR (LENGTH<=12)`, la fonction `LENGTH` est appelée avec l'attribut `LPARZ` comme argument, équivalent à `LENGTH(LPARZ)`.