HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Travail de Bachelor

Développement d'une application web collaborative utilisant la programmation réactive-fonctionnelle

Étudiant
Bastien Clément

Professeur
Marcel Graf

Expert
Julien Richard-Foy

27 juillet 2017

Résumé

Les pages web modernes ne sont plus de simples documents statiques que l'utilisateur se contente de consulter. Ce sont à présent de véritables applications et des plates-formes de collaboration n'ayant plus rien à envier à l'environnement *desktop* classique. La mise à jour en temps-réel du contenu présenté par l'interface de l'application n'est plus un luxe, mais la norme. Ceci est indispensable pour permettre une collaboration efficace entre ses utilisateurs.

Cette tendance se retrouve dans le développement continuel des standards web et des navigateurs. *WebSocket*, *ECMAScript 2015*, *Web Components* sont des exemples des nombreux outils à la disposition du développeur pour bâtir des applications plus dynamiques et modulaires que jamais.

Ce travail explore l'utilisation combinée de ces technologies avec le paradigme de programmation réactive-fonctionnelle. Le concept de *signal*, une valeur pouvant varier avec le temps et offrant un mécanisme de propagation des mises à jour, est introduit puis implémenté sous la forme d'une bibliothèque réutilisable en langage *Scala*.

Cette bibliothèque propose également un ensemble d'outils permettant la définition, en *Scala*, de nouveaux éléments HTML indépendants, composables et conformes aux standards les plus récents. Un système de *data-binding* permet d'injecter la valeur d'un signal directement dans un *template* prédéfini par l'élément. La mise à jour de l'interface est alors gérée de façon automatique par le framework.

Finalement, une application de démonstration est développée et permet d'illustrer l'utilisation conjointe de ces différents concepts. Le compilateur *Scala.js* est utilisé pour permettre l'exécution du code *Scala* directement dans le navigateur.

Cahier des charges

Ce travail de Bachelor a pour objectif de concevoir une application de gestion collaborative qui permet à ses utilisateurs d'interagir en temps réel pour accomplir certaines tâches (organisation du groupe, gestion des membres, gestion des scores, etc.)

1. Conception d'un modèle de données
2. Conception d'une architecture et d'un protocole pour la propagation des données en temps réel entre serveur et clients
3. Conception d'une couche de présentation qui reflète en temps réel le modèle de données dans un document HTML
4. Développement d'une application web (client et serveur) avec programmation en paradigme réactif et fonctionnel (Scala, Scala.js)

Dans la conception l'accent sera mis sur le développement de structures réutilisables dans d'autres applications collaboratives.

Table des matières

Chapitre 1

Introduction

De plus en plus d'activités sociales sont supportées par des technologies de l'information. Les applications web deviennent des médiateurs d'interactions sociales entre personnes, comme par exemple les réseaux sociaux. Des groupes de personnes se rencontrent dans le cyberespace pour s'échanger et poursuivre leurs loisirs. Ces interactions ne sont plus seulement asynchrones, mais deviennent synchrones, par exemple dans le cas des jeux vidéo en ligne. Des groupes de personnes relativement importants se réunissent pour jouer ensemble. Certains jeux nécessitent un haut degré de collaboration entre les membres d'un groupe. Cette collaboration peut être facilité par une application de gestion collaborative.

Les technologies web évoluent très rapidement coté client (navigateur) et permettent de développer des applications de plus en plus interactives et temps réel. Les frameworks récents, comme ReactiveX notamment, offrent un nouveau paradigme de programmation réactive qui simplifie considérablement le développement d'applications hautement interactives.

1.1 Programmation réactive-fonctionnelle

La programmation réactive-fonctionnelle combine programmation fonctionnelle et programmation réactive.

Ce premier paradigme, fonctionnel, met en avant l'utilisation de structures de données immutables et de l'application de fonctions sur ces structures afin d'en dériver de nouvelles. Les effets de bords et les états globaux sont évités par l'utilisation de fonctions pures ne dépendant que de leurs paramètres. Les principes de composabilité des fonctions et de transparence référentielle qui en découlent sont alors mis en avant pour promouvoir un style de pro-

grammation qui se veut plus clair et avec moins d'effets surprenants pour le développeur.

La programmation réactive, quant à elle, prône une approche basée en premier lieu sur les données sources utilisées par l'application, les transformations qui leur sont appliquées et le maintient d'un état cohérent dans l'ensemble du logiciel lors des changements d'états du système.

Dans l'approche mixte réactive-fonctionnelle, les sources de données réactives sont manipulées comme des structures fonctionnelle classiques et des fonctions, pures, leur sont appliquées afin de produire de nouvelles données réactives dérivées. Une relation de dépendance existe entre la donnée réactive produite en résultat de la fonction et la donnée source utilisée. L'objectif du système réactif-fonctionnel est alors de garantir qu'en cas de modification de la donnée source, les données dérivées soient correctement mises à jour par une réapplication de la fonction utilisée en premier lieu.

1.2 Objectifs

Ce travail a pour objectif le développement d'une application web de démonstration, hautement interactive, en utilisant le langage de programmation Scala et les principes de la programmation réactive-fonctionnelle avec une gestion précises et temps-réel des interactions entre ses utilisateurs.

Dans cette optique, une bibliothèque réactive-fonctionnelle – Xuen – sera développée en premier lieu afin de fournir les composants de base utilisés dans la construction de l'application : *signaux*, *templates* et *data-binding*. L'accent sera mis sur le développement d'une bibliothèque générique et réutilisable, indépendante de l'application finale.

1.3 Technologies utilisées

1.3.1 Web Components

Web Components est un ensemble d'extensions aux spécifications HTML et DOM destinées à permettre le développement de composants isolés et réutilisables pour la construction de pages web, de façon simple et portable grâce au support natif de la technologie dans les navigateur modernes.

Le nom *Web Components* regroupe en réalité un ensemble de quatre technologies distinctes, dont le support et la qualité de l'implémentation varie selon le navigateur utilisé :

1. Templates
2. HTML Imports
3. Custom Elements
4. Shadow DOM

Le concept est généralement attribué à Alex Russell [1] qui le présente à l'occasion de la *Fronteers Conference 2011*. Il est alors un développeur de la bibliothèque d'interface Dojo avant d'être engagé par Google pour travailler au développement de la plateforme web, du moteur de rendu Blink et du navigateur Google Chrome [2].

La première implémentation est développée en 2013 par Google comme fondations pour sa bibliothèque *Polymer* [10], principalement dans un but expérimental et afin de collecter des avis. Les autres développeurs de navigateur, notamment Mozilla, Microsoft et Apple n'y participent pratiquement pas et le résultat ne fait pas l'unanimité [3]. De nombreuses discussions sont alors entreprises pour trouver un terrain d'entente et arriver à une spécification finale commune.

État actuel de l'implémentation

Aujourd'hui, les efforts d'implémentation sont toujours en cours. Les dernières versions des navigateurs Chrome, Opera et Safari supportent la plus grande partie de la spécification. L'implémentation de Firefox est toujours en cours de développement tandis que Microsoft considère son implémentation comme *haute-priorité*. La table 1.1 résume l'état actuel des implémentations.

Table 1.1 – Support des technologies Web Component

	Chrome/Opera	Firefox	Safari	Edge
Templates	Stable	Stable	Stable	Partiel
HTML Imports	Stable	Non prévu	Non prévu	Considéré
Custom Elements	Partiel	En cours	Partiel	Considéré
Shadow DOM	Stable	En cours	Buggé	Considéré

Concernant *Custom Elements* et *Shadow DOM*, les versions initiales de ces technologies sont à présents appelées *versions 0*. Les versions actuelles dont il est question ici sont les *versions 1*.

L'implémentation des *Custom Elements v1* par Chrome, Opera et Safari est complète en ce qui concerne la définition de nouveaux éléments indépendants. Il n'est cependant pas possible pour le moment d'étendre des éléments natifs du navigateur, fonctionnalité prévue par la spécification. Dans

le cadre de ce travail, cette fonctionnalité n'étant pas utilisée, il est possible de considérer ces implémentations comme complètes.

Les bugs mentionnés pour l'implémentation de *Shadow DOM v1* par Safari concernent des problèmes liés aux sélecteurs CSS `:host` et `::slotted` introduit par la spécification.

Templates

Cette extension ajoute la balise `<template>` au standard HTML. Son objectif est d'offrir aux développeurs un mécanisme permettant l'inclusion de fragments de page dans un document HTML sans que celui-ci ne soit exécuté, il est alors dit *inerte*.

Comme exemple, sans cet élément, une image avec un attribut `src` était immédiatement téléchargée par le navigateur. Dans une application web utilisant le framework Angular 1, il était alors fréquent de voir des images déclarées en utilisant l'attribut spécial `ng-src` qui était ensuite transformé en simple attribut `src` une fois l'évaluation de l'expression de *databinding* par le framework. Sans cela, le navigateur aurait tenté de charger une image à l'adresse "`{{imgSrc}}`", résultant en une erreur avant que le framework Angular puisse avoir l'occasion d'évaluer l'expression et de modifier l'attribut.

Avec l'élément `<template>`, son contenu est *parsé* et transformé en structure de noeuds DOM de façon similaire à n'importe quel autre élément, mais leurs effets ne sont pas exécutés : les images ne sont pas chargées, les scripts pas exécutés, les styles pas appliqués, etc.

Le contenu de cet élément est ensuite cloné de façon dynamique au *runtime* par un script, puis inséré sous un autre noeud du document en utilisant les APIs du DOM. Ce contenu sera alors exécuté de façon habituelle. Un même template peut également être cloné et inséré plusieurs fois, permettant la factorisation de code répété à travers un document HTML.

HTML Imports

Cette spécification introduit la possibilité pour une page web en HTML d'inclure un second document HTML. Ceci se traduisant par un attribut `rel="import"` sur l'élément `<link>`, déjà utilisé pour lier une feuille de styles CSS à un document.

L'objectif visé est la possibilité pour un composant développé avec les API Web Components d'être contenu dans un unique fichier `.html` et chargé par la page hôte sans recourir à JavaScript.

En pratique, cette fonctionnalité ne semble pas convaincre Mozilla et Apple qui n'ont pour l'instant pas annoncé une intention de l'intégrer dans leurs navigateurs Firefox et Safari.

Dans le cadre de ce travail, cette partie des Web Components ne sera pas utilisée. En effet, l'objectif est de permettre de définir l'ensemble du composant à partir de code Scala compilé par Scala.js. Ce qui signifie combiner au niveau du code source la définition du comportement du composant (en Scala) avec son template (HTML) et son style visuel (CSS). Ainsi, l'ensemble des composants définis sont chargés d'un coup, en même temps que le code JavaScript produit par Scala.js.

Custom Elements

Cette technologie introduit une API pour la définition de nouveau élément DOM. En pratique, cela se traduit par la capacité à créer de nouvelles balises qui seront reconnue par le navigateur et d'y associer des comportements spécifiques lors de son instanciation, attachement ou détachement du document.

Les réponses du composants aux interactions de l'utilisateur se font quant à elles sur la base des événements classiques disponibles en JavaScript : par exemple enregistrement d'un gestionnaire pour l'événement `click` ou `keydown` sur l'élément, tel qu'il serait fait pour un simple élément `<div>`.

La spécification prévoit également la possibilité d'étendre une balise existante selon une forme d'héritage. Il est ainsi possible de définir un type de bouton personnalisé, qui hériterait du comportement standard d'un bouton mais qu'il serait alors possible d'altérer. En pratique, cette fonctionnalité n'est pas encore supportée par les navigateurs.

La définition de nouvelle balise est un concept clé de ce travail puisque chaque composant créé en Scala.js sera instancié par l'utilisation de sa balise associée dans le template d'un autre élément. En revanche, l'extension d'éléments existants ne sera pas exploité. Il est en effet trivial d'atteindre un effet similaire en définissant un nouvel élément contenant l'élément à altérer dans son sous-arbre DOM.

Shadow DOM

Probablement la spécification la plus complexe d'un point de vue implémentation pour les navigateurs et la plus difficile à *polyfill* en JavaScript. Elle définit un mécanisme permettant d'encapsuler un sous-arbre DOM et des styles CSS associés et de les rendre invisibles de l'extérieur. Ainsi :

— les styles CSS déclarés hors du sous-arbre ne s'appliquent pas aux éléments du sous-arbre et inversement pour les styles déclarés dans le sous-arbre ;

— les identifiants HTML (attributs `id`), en principe uniques dans un document, peuvent être réutilisés dans un sous-arbre ;

— les fonctions `getElementById` ou `querySelector` dépendent de l'élément depuis lequel elles sont appelées et ne peuvent pas traverser une frontière formée par un sous-arbre ;

— les événements sont *retargeted* à leur sortie d'un sous-arbre, leur attribut `target` indiquant depuis quel élément ils ont été émis est modifié pour référencer le noeud à la racine du sous-arbre au lieu de l'élément qui a réellement émis l'événement, à l'intérieur du sous-arbre.

Ce mécanisme est également clé à la réalisation de ce projet puisque chaque élément sera associé à son sous-arbre DOM caché contenant une instance de son template.

1.3.2 Scala.js

Scala.js est un plugin du compilateur Scala produisant du code JavaScript en sortie à la place du bytecode Java. Ce code est alors directement interprétable par un navigateur web standard et ne dépend plus de la machine virtuelle Java (JVM). Scala.js peut également être utilisé avec d'autres plateforme basée sur JavaScript, par exemple Node.js.

Conceptuellement similaire à de nombreux autre projets tels que TypeScript [18], ClojureScript [19] et même Emscripten [20] comme approche alternative au développement pour le web, Scala.js se distingue par le support complet du langage Scala, y compris de ses fonctionnalités les plus avancées, ainsi que par une excellente interopérabilité avec les interfaces JavaScript natives du navigateur.

Scala.js est aussi une option intéressante lorsque le langage Scala est utilisé pour le développement du *backend* d'une application web. Permettant alors le partage de sources entre le développement côté client et côté serveur. Il est alors fréquent de mettre en place un projet commun entre le serveur et le client afin de partager les définitions des données échangées sur le réseau. La communication entre les deux parties de l'application s'en trouve alors considérablement simplifiée puisque les deux *parlent* le même langage.

Une attention particulière a été portée à respecter la sémantique d'exécution de la JVM, les différences entre les plateformes sont généralement invisibles pour le développeur et un code source identique s'exécute correctement et sans modification sur les deux plateformes.

Le projet est à présent bien intégré dans l'écosystème Scala : de nombreuses bibliothèques sont *cross-compilées* à la fois pour la plateforme Java et pour JavaScript et le projet est considéré stable pour une utilisation en production depuis plusieurs années. Le support de multiples plateformes d'exécution pour le langage Scala est renforcé par le développement récent de Scala Native [16] visant la génération de code natif à partir de sources Scala en utilisant le compilateur LLVM [17].

1.3.3 Limitations actuelles

Les *Web Components* sont une technologie *cutting edge*. Très récemment standardisée, son support par les différents navigateurs laisse encore beaucoup à désirer. De plus, la spécification bâtit sur les bases du langage *ECMAScript 2015* (ES6), la dernière version du langage JavaScript. Il est par exemple nécessaire de déclarer une *classe*[1] JavaScript pour implémenter le comportement d'un élément personnalisé, par opposition à une simple fonction constructeur tel que c'était le cas dans la version antérieure du standard. L'utilisation d'ES6 devient donc obligatoire, malgré que son support actuel soit toujours très variable entre les différents navigateurs.

Heureusement, le compilateur Scala.js possède une option lui indiquant de produire du code JavaScript ES6 plutôt que la version plus répandue ES5. Lorsque cette option est activée, le compilateur utilise extensivement les nouveaux mécanismes du langage ES6, notamment les classes, et permet la définition d'éléments personnalisés conformément aux standards.

Cette option introduit cependant une série de complications :

— Le mode ES6 de Scala.js est encore considéré comme expérimental. En pratique, la génération de code ne semble pas être un problème et le code produit s'exécute parfaitement avec le navigateur Google Chrome. En revanche, Firefox semble avoir quelques problèmes avec des structures pourtant standards.

— L'optimisateur de Scala.js est construit sur la base du *Closure Compiler* [21], un compilateur *Javascript-to-Javascript* de Google dont la tâche est d'optimiser et compresser le code JavaScript traité. Cependant, celui-ci ne supporte pas encore ES6. Le compilateur Scala.js tend à produire des noms de fonctions et variables particulièrement longs, ce qui gonfle rapidement le taille du fichier final si l'optimisateur ne peut pas effectuer son travail.

— L'option n'affecte que la génération finale du fichier JavaScript. Lorsqu'une bibliothèque Scala.js est publiée, celle-ci contient à la place

1. Construction introduite avec la version ES6

des fichiers *.sjsir*, une représentation intermédiaire pour le compilateur Scala.js. Ces fichiers seront alors utilisés pour construire l'unique fichier *.js* lors du processus de compilation de l'application finale. Il est ainsi nécessaire que le support d'ES6 soit activé explicitement par l'utilisateur de la bibliothèque, ce qui peut compliquer légèrement son intégration dans un projet existant.

Au final, le choix a été fait d'ignorer ces complications. ES6 est la version actuelle du langage JavaScript et son support est un effort important du développement des navigateurs. Le fait que les spécifications le plus récentes l'imposent indique qu'il est temps pour les navigateurs et les développeurs d'évoluer vers le nouveau standard.

Le support d'ES6 par *Closure Compiler* est également un domaine de développement actif du projet. D'ici là, l'absence de l'optimisateur peut être compensée par l'utilisation de la compression au niveau du protocole HTTP. En mode ES5, une compression *GZip* sur le fichier original sans optimisation produit en effet un fichier plus léger que la version produite par l'optimisateur [2]. L'idéal reste bien sûr la combinaison des deux, mais le résultat est déjà plus que satisfaisant.

Du fait de ces différentes limitations et du support actuel des technologies *Web Components*, une application construite avec Xuen ne peut être exécutée que dans un navigateur basé sur les moteurs de rendu *Blink* ou *WebKit*, c'est à dire Google Chrome, Opera et Safari. Les navigateurs Firefox et Edge ne supportent actuellement pas les *Web Components*.

En pratique, Safari présente d'autres limitations au niveau des technologies CSS utilisées pour la construction de l'interface de l'application de démonstration, notamment au niveau du support *FlexBox* [22] et *CSS Containment* [23]. Le développement a donc été focalisé sur la plateforme Blink.

À plus long terme, l'évolution des navigateurs et des outils utilisés pour s'aligner avec les standards actuels devrait permettre l'utilisation de la bibliothèque de façon optimale et sans complications sur tous les navigateurs, avec un minimum d'adaptations.

2. Source : 1779KB, Opt : 389KB, Gzip : 190KB, Opt + Gzip : 83KB

Première partie

Bibliothèque réactive-fonctionnelle

Xuen

Chapitre 2

Signaux

2.1 Motivations

La construction d'interfaces utilisateur met en évidence la problématique de la gestion des interactions et du maintien de la cohérence des informations présentées. En effet, les actions effectuées par l'utilisateur modifient l'état du logiciel et requièrent alors une actualisation de l'affichage. Lorsque l'interface devient complexe, maintenir une cohérence globale présente une difficulté de plus en plus importante. Le problème est exacerbé lorsque les modifications de l'état ne proviennent pas uniquement de l'utilisateur mais peuvent également survenir par l'action de processus asynchrones tel qu'une tâche de fond ou une connexion réseau.

La séparation classique Modèle-Vue-Contrôleur utilise généralement la notion d'*Observable* et d'*Observateur* pour lier Vue et Modèle. Ce concept présente cependant de multiples inconvénients tels que la promotion d'effets de bord, une diminution de l'encapsulation, une verbosité excessive ; le rendant ainsi fastidieux à l'utilisation et sujet à erreurs [4].

Ingo Maier et Martin Odersky proposent ainsi une approche plus fonctionnelle et composable avec la bibliothèque *Scala.React* [5] avec, entre autres, la notion de signal : une valeur pouvant varier avec le temps. Cependant les signaux ne sont qu'un des multiples outils mis à disposition et l'utilisation de la bibliothèque se révèle être excessivement complexe, même dans les cas les plus simples [6, Related Work]. Cette complexité découle d'une implémentation soucieuse des difficultés introduites par un environnement multi-thread, fréquent lors du développement d'application de type *fat-client* s'exécutant sur la machine virtuelle Java.

Partant de ce constat, Li Haoyi a ainsi développé *Scala.Rx* [6] : une réimplémentation simplifiée du concept de signaux avec une emphase sur la

simplicité, à la fois au niveau de la conception que celui de l'utilisation. Cependant, par simplicité justement, plusieurs limitations ont été volontairement imposées et se révèlent être particulièrement gênantes dans le cadre de ce projet (§ 2.3.2).

Xuen implémente ainsi un concept de signaux largement basés sur ceux de *Scala.Rx*, mais dont les fonctionnalités ont été spécifiquement adaptées à leur utilisation dans le cadre du développement d'interfaces utilisateur et de l'utilisation de données chargées de façon asynchrone.

2.2 Spécifications

2.2.1 Définition

Un `Signal[T]` représente une information de type `T` dont la disponibilité ou la valeur peut varier avec le temps. À tout moment, il peut se trouver dans l'un des deux états suivants :

1. `Undefined` : le signal ne possède pas de valeur définie,
2. `Defined(value)` : le signal possède actuellement la valeur `value`

Il peut être vu comme une extension de `Future[T]`. De façon similaire, il représente la présence ou l'absence d'information au fil du temps, mais à la différence de `Future`, il est autorisé à changer d'état un nombre arbitraire de fois tandis que l'état d'un `Future` est figé une fois celui-ci résolu.

Le type `Signal[T]` est covariant avec son paramètre `T`. L'interface exposée ne permettant que l'accès à la valeur du signal ou sa transformation par le biais de la construction d'un nouveau signal, une instance `Signal[B]` est substituable à `Signal[A]` si `B <: A`[1].

L'état d'un signal peut être dépendant de l'état d'un ou plusieurs autres signaux. Il constitue alors un signal *enfant* associé à un ensemble de signaux *parents*. Cet ensemble peut varier dynamiquement au fil du temps. Un signal enfant ne peut changer d'état que lorsque au moins l'un des ses signaux parents change d'état.

À l'inverse, un signal qui ne dépend d'aucun autre est appelé une *source*. Le changement d'état d'un signal source ne peut s'opérer que par une mutation explicite, extérieur au système de signaux.

1. `B <: A` désigne une relation de sous-type (`B extends A`)

`Signal` vérifie les trois axiomes des monades [7] :

$$\begin{aligned} Signal(x) \text{ flatMap } f &\equiv f(x) \\ a \text{ flatMap } (x \mapsto Signal(x)) &\equiv a \\ (a \text{ flatMap } f) \text{ flatMap } g &\equiv a \text{ flatMap } (x \mapsto f(x) \text{ flatMap } g) \end{aligned}$$

Ici, l'opérateur $\equiv$ désigne une équivalence structurelle : les deux expressions sont substituables sans affecter le comportement du programme.

2.2.2 Accès à l'état courant

L'interface d'un signal `Signal[T]` définit deux méthodes pour accéder à sa valeur courante :

1. **`Signal.option`** : retourne la valeur courante d'un signal sous la forme d'une `Option[T]`. C'est une façon sûre d'accéder à l'état du signal quel qu'il soit.
2. **`Signal.value`** : retourne la valeur courante du signal (donc une valeur de type `T`) s'il est défini ou lance une exception [2] s'il ne l'est pas. De façon générale, cette méthode est plutôt destinée à être utilisée dans le cadre de la définition de signaux expression (section 2.2.7) puisque, dans ce cas, l'exception est traitée par le constructeur et entraîne la construction d'un signal vide.

Dans le cas où il est nécessaire d'être informé des futurs changements d'états du signal, le mécanisme d'observateur (§ 2.2.9) peut être utilisé.

2.2.3 Pureté et modes d'évaluation

Un signal est considéré comme une construction fonctionnelle semi-pure : il n'est dépendant d'aucun état global à l'exception d'autres signaux et ne présente pas d'effets de bords. Ces contraintes s'étendent également aux fonctions utilisées dans leur définition ou transformation par le biais de méthodes telles que `map` ou `filter`. Le langage Scala ne permettant pas d'imposer la notion de pureté aux fonctions, il est de la responsabilité du développeur de s'assurer que cette contrainte soit respectée.

Cette contrainte découle principalement de l'existence de deux modes d'évaluation pour les signaux qui définissent à quel moment l'état d'un signal enfant doit être calculé après le changement d'état d'un de ses signaux parent :

2. De type `UndefinedSignalException`

1. *Paresseux* : l'état du signal enfant est réévalué à la demande, lorsque son état courant est accédé et après un changement d'état d'au moins un des ses signaux parents,
2. *Différé* : l'état du signal enfant est réévalué à la fermeture d'un contexte de mutation impliquant la modification d'au moins un signal parent.

Le mode d'évaluation paresseux est le mode utilisé par défaut pour les signaux. Il offre l'avantage de réduire le nombre d'états à calculer dans le cas où un signal n'est pas accédé aussi fréquemment que son état ne change. Il n'offre cependant aucune garantie concernant le moment auquel une fonction utilisée comme définition d'un signal ou d'une transformation sera appelée, ni même qu'elle sera un jour appelée. Il est donc particulièrement important que ces fonctions respectent également la contrainte de pureté des signaux pour ne pas introduire des comportements non-déterministes dans l'application.

Le mode différé n'est utilisé que dans les situations où l'événement de changement d'état est significatif en plus de la valeur de ce nouvel état. C'est par exemple le cas de l'opération `fold` (§ 2.2.8) qui est utilisée pour combiner les états successifs d'un signal parent afin de construire l'état du signal enfant. Dans une telle situation, l'utilisation du mode paresseux rendrait le signal enfant non-déterministe puisque son état serait dépendant de la fréquence à laquelle il est accédé [3].

Par ailleurs, entre les changements d'états d'un signal parent, l'état d'un signal enfant est gardé en mémoire. L'objectif est d'éviter de devoir recalculer l'ensemble d'une hiérarchie de signaux à chaque accès d'un signal enfant. La valeur d'un signal enfant n'étant pas sensé changer sans un changement d'état d'un parent.

En résumé, l'usage combiné de la *memoization* et de l'évaluation paresseuse rend le timing d'évaluation des fonctions utilisées pour la définition et la transformation de signaux imprévisible, d'autant plus lorsque l'application et la structure de graphe de signaux se complexifient. Respecter la contrainte de pureté lors de l'utilisation des signaux permet de s'affranchir de cette complexité et de se concentrer sur la logique métier de l'application développée.

2.2.4 Hiérarchie simplifiée

La figure 2.1 présente une version simplifiée de la hiérarchie des signaux. Seules les informations pertinentes à l'utilisation des signaux ont été conser-

3. Si la fréquence d'accès est inférieure à la fréquence de changement d'état du signal parent, certains états seront *manqués* par l'opération de combinaison.

vée : les méthodes, interfaces, classes intermédiaires relevant des détails d'implémentation ne sont pas incluses. La hiérarchie complète formée par les signaux est présentée en section 2.4.1.

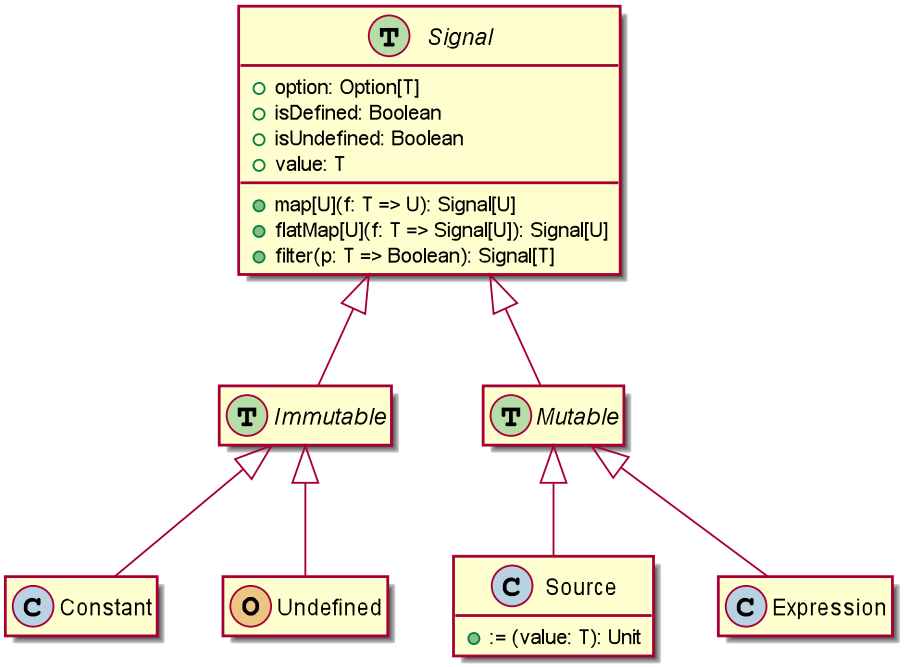

FIGURE 2.1 – Hiérarchie simplifiée des signaux

Le trait racine `Signal` est l'interface générique destinée à être manipulée par le développeur, quelque soit le type concret de signal manipulé. Il expose les méthodes nécessaires à l'accès à l'état courant du signal ainsi qu'à la construction de signaux dérivés par transformation. C'est une interface immutable qui ne permet de modifier l'état courant du signal.

La séparation des signaux en deux sous-arbres, `Mutable` et `Immutable`, capture les différences sémantiques liées à la présence ou l'absence d'une garantie d'immutabilité de l'état d'un signal.

L'intérêt d'un signal dont la valeur ne varie pas peut sembler limité a priori mais se révèle lorsqu'une valeur non-signal doit être encapsulée dans un signal afin de satisfaire le système de types. Un tel signal n'a pas de raison de changer d'état au fil du temps, il est alors effectivement immutable. La présence de cette contrainte de façon explicite au niveau de la hiérarchie permet la mise en place d'un certain nombre d'optimisations.

Une instance de la classe `Constant[T]` est simplement une valeur de type `T` avec une interface de signal. Un tel signal est toujours dans l'état *défini*.

L'objet singleton `Undefined` représente quant à lui le signal dont l'état est toujours *indéfini*. Il est généralement référencé à partir de l'objet compagnon du trait `Signal` : sous la forme `Signal.undefined`. Puisqu'il n'est jamais défini, il est déclaré comme une instance de `Signal[Nothing]`. Il est ainsi, par covariance, substituable à n'importe quel type de signal `Signal[T]` [4].

Un signal immutable ne peut être ni *enfant* ni *parent* d'autres signaux. Comme ils ne changent jamais, il est inutile de maintenir de graphe de dépendances entre eux. Il n'y aura en effet jamais de propagation de changement d'état à effectuer.

Les deux types de signaux mutables, sources (§ 2.2.6) et expressions (§ 2.2.7), sont décrits plus en détails dans les sections les concernant.

2.2.5 Construction

Un signal est généralement dérivé par transformation de signaux existants. Cependant, dans le cas où un nouveau signal racine doit être construit, deux approches sont disponibles.

La première méthode est l'utilisation des constructeurs offerts par l'objet compagnon du trait `Signal` :

— `def Signal.apply[T](expr: =>T): Signal[T]`

— `def Signal.define[T](expr: =>Option[T]): Signal[T]`

L'expression fournie à `apply` sera évaluée pour déterminer la valeur du signal produit et les dépendances vers d'autres signaux seront automatiquement identifiées. Si l'expression fait référence à l'état d'au moins un signal parent non-constant, un signal de type `Expression` (§ 2.2.7) est retourné. Dans le cas où aucun signal parent n'a été accédé, un signal de type `Constant` est retourné.

La variante `define` est similaire mais considère une valeur `None` comme un signal indéfini tandis qu'une valeur `Some(v)` est considérée comme un signal défini et de valeur `v`.

S'il n'est pas souhaitable que l'expression de définition soit évaluée immédiatement, il est possible de différer son évaluation au premier accès à l'état du signal produit en utilisant le constructeur

```
def Signal.defer[T](expr: =>T): Signal[T]
```

4. `Nothing` est le *bottom type* du système de types en Scala, il est sous-type de tous les types ($\forall$`T`, `Nothing <: T`) mais il n'en existe aucune instance.

Le signal ainsi produit sera de type `Expression[T]`, même si l'expression passée ne fait référence à aucun autre signal, formant alors un signal expression effectivement constant.

Les méthodes `apply`, `define` et `defer` prennent en réalité des paramètres optionnels supplémentaires permettant de configurer la sémantique du signal défini. La signature complète de ces méthodes est disponible dans la Scaladoc du projet.

La seconde approche consiste en la construction explicite d'un signal de type `Source` (§ 2.2.6).

2.2.6 Signal source

Une `Source[T]` est l'équivalent réactif d'une variable en programmation non-réactive. C'est un conteneur mutable pour une valeur de type `T`, pouvant être indéfinie. Deux constructeurs sont disponibles selon l'état initial désiré pour la source :

— `def Source.apply[T](value: T): Source[T]`

— `def Source.undefined[T]: Source[T]`

Une instance de `Source[T]` offre une méthode de mutation explicite

```
def :=(value: T): Unit
```

permettant de mettre à jour la valeur contenue dans la source de façon impérative. Cette opération est un changement d'état de la source et provoquera l'invalidation récursive des tous les signaux qui en dépendent.

Une source est destinée à être utilisées lors de la construction de système hybrides, combinant code impératif basé sur les effets de bords et code fonctionnel. La source est alors un point d'entrée dans le graphe de dépendances des signaux pour la partie de code impérative.

Une surcharge de l'opérateur de mutation

```
def :=(nil: Signal.nil.type): Unit
```

permet de définir explicitement l'état indéfini pour la source. Cette surcharge est invoquée en passant explicitement l'objet `Signal.nil` en tant qu'opérande de droite à l'opérateur `:=`.

Finalement, si la nouvelle valeur d'une source est dérivée de sa valeur actuelle, l'opérateur de mise à jour

```
def ~=(f: T =>T): Unit
```

permet de fournir une fonction qui sera invoquée avec la valeur courante de la source et dont la valeur de retour sera utilisée comme nouvelle valeur de la source. Si la source est actuellement indéfinie, l'opérateur n'a aucun effet et la fonction n'est pas invoquée.

Cet opérateur est plus qu'un simple sucre syntaxique par rapport à l'usage explicite des méthodes `option` et `value`. Dans le contexte des observateurs (§ 2.2.9), accéder à la valeur d'une source (de même que n'importe quel type de signal mutable) établit une dépendance entre le signal et l'observateur. L'observateur est donc invoqué à chaque changement de la source et son implémentation consiste en la mutation de cette même source, établissant ainsi une boucle infinie.

À l'inverse, l'opérateur `~=` ne construit pas de dépendance entre la source et l'observateur. Il est bien entendu de la responsabilité du développeur de s'assurer de ne pas *leak* la valeur précédente ou courante de cette source hors de la fonction de mutation.

2.2.7 Signal expression

Un signal expression est un signal dont la définition est une expression arbitraire. Un tel signal détermine automatiquement ses signaux parents en observant les signaux accédés lors de l'évaluation de l'expression et construit ainsi automatiquement son arbre de dépendances. Si l'un de ces signaux venait à changer, la valeur du signal expression serait recalculée.

Il est construit en passant l'expression de définition au constructeur `Signal` tel qu'illustré par la figure 2.2. La variante `Signal.define` est similaire à la méthode `apply`, mais reçoit une expression de type `Option[T]`. Une évaluation de cette expression produisant la valeur `None` ou une instance `Some(v)` conduit respectivement à un état indéfini ou défini du signal.

```
1 val a: Signal[Int] = ...
2 val b: Signal[Int] = ...
3 val c: Signal[Int] = Signal {
4   a.value + b.value
5 }
```

FIGURE 2.2 – Déclaration d'un signal expression

Les parents d'un signal expression sont dynamiques. À chaque évaluation, la liste des parents est vidée puis reconstruite selon l'évaluation actuelle. De cette façon, les dépendances sont toujours les plus précises possible et les invalidations inutiles sont évitées. Ceci est particulièrement important dans le

cas de signaux contenant des branches et donc un ensemble de dépendances dynamiques selon l'état d'autres signaux.

Dans l'exemple de la figure 2.3, le signal construit ne dépend de `b` que si la valeur du signal `a` est `false`. Dans le cas contraire, il est dépendant de `c`. Dans tous les cas, une dépendance est créée vers le signal `a`.

```
1 Signal {
2   if (a.value) b.value else c.value
3 }
```

FIGURE 2.3 – Définition d'un signal expression avec branches

Selon la situation, l'usage d'une expression pour définir un signal peut se révéler plus simple que la combinaison de nombreuses opérations de transformations élémentaires pour composer le comportement attendu.

Dans les cas les plus complexes, principalement lors de l'utilisation de structures de contrôles tel que des branches conditionnelles, des opérations de *pattern matching* ou des boucles, une expression permet une définition concise et atomique du signal tandis que pour obtenir un résultat équivalent à l'aide des opérateurs de transformation, une longue chaîne de transformations successives, considérablement plus difficile à comprendre, serait nécessaire.

À l'inverse, dans les cas plus simples, une opération de transformation permet de réutiliser un *pattern* de transformation établi, étant alors à la fois plus concis et mentalement plus simple puisqu'il utilise une sémantique clairement établie et commune.

Contraintes des expressions

Les signaux doivent être considérés comme des constructions semi-pures d'un point de vue fonctionnel (§ 2.2.3). Il est ainsi important que l'expression utilisée comme définition respecte ce principe en ne provoquant aucun effet de bord et en ne dépendant d'aucune valeur mutable qui ne serait pas un signal. En effet, si une valeur mutable non-signal est référencée par une expression, l'état résultant du signal est alors dépendant de l'instant d'évaluation pour lequel aucune garantie n'est fournie.

Il est aussi important que l'évaluation d'une expression s'effectue de façon synchrone. En effet la liste de dépendances du signal est construite lors de l'évaluation de l'expression. Si un signal parent est accédé de façon asynchrone ou *lazy*, la dépendance ne sera pas identifiée et le signal ne sera pas correctement invalidé en cas de changement d'état de ce signal parent.

Les deux principaux suspects à considérer sont `Future` et `Stream`. Le premier pour le délai qu'il introduit dans l'évaluation de sa valeur, le second pour sa sémantique *lazy*.

Il est intéressant de noter que la seule observation du paramètre de type du signal permet d'identifier un potentiel problème. En effet un signal de type `Signal[Int]` dont la définition impliquerait l'utilisation d'une instance de `Stream` n'est pas un souci ; une fois la valeur finale de type `Int` produite, l'ensemble des éléments pertinents du flux auront été consommés de façon synchrone. Le principe s'applique de façon similaire à l'opération `Option.orElse` pour laquelle le paramètre est passé *by name*.

À l'inverse, un signal `Signal[Stream[Int]]` expose l'instance de `Stream` utilisée. Dans une telle situation, si le calcul d'un élément du flux requiert l'accès à un autre signal, aucune relation de dépendance ne sera établie. Des types de signaux tels que `Signal[Stream[A => B]]`, `Signal[Future[A]]` voir même `Signal[A => B]` indiquent un risque important ne pas respecter la contrainte de synchronisme.

2.2.8 Opérateurs de transformations

Dans les exemples ci-dessous, les opérations sont supposées être appliquées à une instance de type `Signal[T]`, `T` faisant ainsi référence au type d'élément contenu dans le signal original. Seules les opérateurs les plus courants et ceux utilisées par le compilateur Scala lors de la compilation d'une compréhension `for` sont traités ici. La Scaladoc du projet contient une liste exhaustive des opérations disponibles.

Ces transformations sont pour la plupart construites autour du constructeur `Signal.define`. Leur évaluation est donc immédiate et produit une valeur de retour en fonction des dépendances du signal original et des signaux accédés par la fonction de transformation fournie. Par exemple, invoquer la méthode `map` sur un signal `Constant` retournera en principe un signal `Constant`, sauf si la fonction passée à `map` fait référence à la valeur d'autres signaux, produisant alors un signal `Expression` non-constant.

Opérateur de transformation simple (`map`)

— `def map[U](f: T=>U): Signal[U]`

La fonction `map` effectue une opération de transformation simple sur la valeur d'un signal en appliquant la fonction `f` à la valeur courante du signal et retournant un nouveau signal contenant en tout temps le résultat de cette transformation. En d'autre termes, lorsque l'état du signal original change,

la fonction `f` est réévaluée avec la nouvelle valeur du signal parent et le signal enfant est mis à jour.

Si le signal d'origine est indéfini, la fonction `f` n'est pas appliquée et le signal enfant prend également l'état indéfini.

```
1 val a: Signal[Int] = Source(4)
2 val b: Signal[Double] = a.map(Math.sqrt(_)) // b.value -> 2.0
3 a := 9 // b.value -> 3.0
4
5 val c: Signal[_] = Signal.undefined // `c` est indéfini
6 val d: Signal[_] = c.map(value => ???)
7 d.option == None // la fonction passée à `map` n'est jamais évaluée
```

FIGURE 2.4 – Exemple d'utilisation de `map`

Opérateur de sélection (`flatMap`)

— `def flatMap[U](f: T=>Signal[U]): Signal[U]`

Dans le cas des signaux, `flatMap` implémente une opération de sélection : étant donné un signal `a` de type `Signal[T]` et une transformation sous la forme d'une fonction `f: T => Signal[U]`, la fonction `f` est appliquée à la valeur actuelle du signal `a` afin d'obtenir un second signal `b` de type `Signal[U]` et retourne un troisième signal `c` également de type `Signal[U]` dont la valeur est en tout temps égale à celle du signal `b`. La fonction `f` est réévaluée à chaque changement d'état du signal `a` afin de définir un nouveau signal de référence `b`.

Si le signal `a` est indéfini, la fonction `f` n'est pas appliquée et le signal `c` est également considéré indéfini.

La fonction `flatMap` [5] est la fonction universelle de transformation des monades. Elle est suffisamment générale pour permettre de définir toutes les autres fonctions de transformation comme des cas particuliers de `flatMap`. Par exemple, la transformation `a.map(f)` peut également s'écrire sous la forme `a.flatMap(value => Constant(f(value))`.

Opérateur de filtrage (`filter`)

— `def filter(p: T=>Boolean): Signal[T]`

5. *bind* en Haskell, ou `>>=`

```
1 val choice: Signal[Int] = ...
2 def selectSignal(choice: Int): Signal[T] = ...
3 // L'état de `c` est identique à celui du signal retourné par
      `selectSignal` pour la valeur courante de `choice`
4 val c: Signal[T] = choice.flatMap(selectSignal)
```

FIGURE 2.5 – Exemple d'utilisation de `flatMap`

La fonction `filter` effectue une opération de filtrage d'un signal en appliquant un prédicat `p` à la valeur courante du signal et retournant un nouveau signal de même valeur si le prédicat est vérifié, ou un signal indéfini si le prédicat n'est pas vérifié.

```
1   val a: Signal[Int] = Source(4)
2   val b: Signal[Int] = a.filter(_ % 2 == 0) // b.option -> Some(4)
3   a := 5 // b.option -> None, b.value -> UndefinedSignalException
```

FIGURE 2.6 – Exemple d'utilisation de `filter`

Opérateur de combinaison (`fold` / `reduce`)

— `def fold[U](a: U)(f: (U, T)=>U): Signal[U]`

— `def reduce[U >:T](f: (U, T)=>U): Signal[U]`

L'opérateur `fold` permet l'introduction d'un effet de *mémoire* aux signaux. Il prend en paramètre un *accumulateur initial* `a` de type `U` et une fonction de combinaison `f: (U, T) => U` permettant d'associer la valeur courante du signal à cet état pour produire un *accumulateur courant*, également de type `U`, qui sera alors la valeur du signal produit par l'opérateur.

Lors d'un changement d'état du signal initial, l'*accumulateur antérieur* est combiné à la nouvelle valeur du signal pour former le nouvel *accumulateur courant*. Si le signal est indéfini, la fonction `f` n'est pas évaluée et l'*accumulateur antérieur* devient l'*accumulateur courant* sans modification. Dans le cas où le signal est initialement indéfini, l'*accumulateur initial* devient l'*accumulateur courant* tel quel.

Le signal retourné par `fold` est toujours défini.

L'opérateur `fold` possède la particularité d'être affecté par le mode d'évaluation du signal qu'il produit. En mode paresseux, l'opérateur de combinaison ne serait appliqué que lors de l'accès au signal enfant. Il serait alors possible de manquer des changements d'état du signal parent. C'est pourquoi les

signaux produits par l'opérateur `fold` ont toujours un mode d'évaluation différé afin d'obtenir un comportement déterministe et indépendant de la façon dont le signal enfant est utilisé.

À l'inverse de la fonction `fold` présente sur les collections de la bibliothèque standard Scala qui retourne une unique valeur pour une collection, la fonction `fold` des signaux retourne également un `Signal`. Le nom *fold* ne fait ainsi pas référence au passage d'une collection à un élément unique, mais à la combinaison successive des différents *états* du signal parent.

```
1 val a: Signal[Int] = ...
2 val s: Signal[Int] = a.fold(0)(_ + _)
3 // Le signal `s` représente la somme de toutes les valeurs du
      signal `a`.
4
5 val b: Signal[Int] = ...
6 val m: Signal[Int] = b.fold(0)(_ max _)
7 // Le signal `m` représente la valeur maximale obtenue par le
      signal `b`.
```

FIGURE 2.7 – Exemple d'utilisation de `fold`

L'opérateur `reduce` est une variation de l'opérateur `fold` dont l'*accumulateur initial* est déterminé implicitement par l'état courant du signal parent. À l'inverse de la transformation `fold`, si l'état courant du signal parent est indéfini, alors le signal enfant est également indéfini. Dès lors que l'état du signal parent sera défini pour la première fois, l'état du signal enfant ne pourra plus être indéfini.

```
1 val a: Signal[Int] = ...
2 val b: Signal[Int] = a.reduce((acc, x) => x)
3 // La valeur du signal `b` reflète la valeur du signal `a` lorsque
      celui-ci est défini. Lorsqu'il est indéfini, le signal `b`
      contient la dernière valeur définie du signal `a`. Si `a` n'a
      jamais été défini, `b` est indéfini.
4
5 val c: Signal[Int] = ...
6 val d: Signal[Int] = c.reduce(_ max _)
7 // Similaire à l'exemple correspondant pour `fold`, mais ne suppose
      pas une valeur intiale de 0. Si `c` représente des nombres
      négatifs, la version avec `fold` ne serait pas correcte (il
      faudrait utiliser Int.MinValue).
```

FIGURE 2.8 – Exemple d'utilisation de `reduce`

Opérateurs d'encapsulation (`wrap / unwrap`)

- `def wrap: Signal[Option[T]]`
- `def unwrap[U](implicit ev: T <:< Option[U]): Signal[U]`

L'opérateur `wrap` transforme un signal d'origine de type `T` en un signal de type `Option[T]`. Lorsque le signal initial est défini, ce nouveau signal sera défini à `Some(value)`, avec `value` la valeur actuelle du signal original. Dans le cas où il serait indéfini, le signal de retour est défini à `None`.

Cette opération garantit ainsi un signal toujours défini à une instance d'`Option` et permet de contourner la sémantique des opérateurs de transformations vis-à-vis des signaux indéfinis. Ceci permet de traiter avec des opérateurs tel que `map`, `fold`, etc., les valeurs définies mais également indéfinies d'un signal.

```
1 // Construction d'un signal avec une valeur par défaut qui sera
      utilisée si le signal original est indéfini
2 def withDefault[U >: T](s: Signal[T]])(default: U): Signal[U] = {
3   val a: Signal[Option[T]] = s.wrap
4   a.map((opt: Option[T]) => opt.getOrElse(default))
5 }
```

FIGURE 2.9 – Exemple d'utilisation de `wrap`

L'opération `unwrap` effectue la transformation inverse. Si le signal initial possède une valeur `Some(v)`, la valeur du signal retourné est simplement égale à `v`. Dans le cas où le signal original vaut `None`, le signal résultant est indéfini. Cette opération ne peut être appliquée que sur une instance de signal de type `Signal[Option[U]]` pour un `U` quelconque.

Le paramètre implicite n'est rien d'autre qu'une implémentation de cette contrainte[6]. Si celle-ci est respectée, le compilateur Scala sera en mesure de fournir une valeur pour ce paramètre implicite. En revanche, si le type du signal ne correspond pas, une erreur de compilation liée à l'absence du paramètre implicite sera générée. Le développeur n'a donc pas à se soucier de ce paramètre implicite et peut se contenter d'utiliser cette méthode tel que si elle ne prenait aucun paramètre.

2.2.9 Observateurs

Les observateurs permettent d'ajouter des effets de bords aux changements d'états d'un signal. De la même façon que les sources sont les points d'entrée

6. La même technique est utilisée par les collections de Scala pour l'implémentation de la méthode `flatten`.

```
// Définition de l'opérateur `reduce` à partir de `fold` et `unwrap`
def reduce[U >: T](s: Signal[T])(op: (U, T) => U): Signal[U] = {
  val a: Signal[Option[U]] =
    s.fold[Option[U]](None) { (prev: Option[U], cur: T) =>
      prev.map(op(_, cur))
    }
  a.unwrap
}
```

FIGURE 2.10 – Exemple d'utilisation de `unwrap`

dans un graphe de signaux, les observateurs sont les points de sorties.

Un observateur se construit par l'invocation du constructeur

`Observer.apply(body: =>Unit): Observer`

Il se comporte de façon similaire à un signal différé. Lorsque l'état d'un signal change, tous les observateurs attachés à ce signal sont placés dans une queue. Le processus se poursuit alors pour les signaux enfants, ajoutant les observateurs attachés à ces signaux à la queue. Une fois l'ensemble des signaux enfants invalidés, les observateurs sont invoqués (§ 2.2.10).

Un observateur peut se trouver dans deux états distinct :

1. Attaché (*bound*) : l'observateur est actif et sera invoqué pour chaque changement d'état des signaux qu'il observe.
2. Détaché (*unbound*) : l'observateur est inactif, il est effectivement retiré de tous les signaux qu'il observait précédemment et ne sera plus invoqué jusqu'à ce qu'il soit explicitement réactivé.

Un observateur peut être attaché ou détaché en invoquant les méthodes `bind()` et `unbind()`. Lorsqu'un observateur transitionne de l'état détaché à l'état attaché, sa fonction de définition est invoquée afin de déterminer les signaux auxquels l'attacher.

Lors de la construction d'un nouvel observateur, celui-ci est initialement *attaché*, impliquant que la fonction utilisée dans sa définition est immédiatement invoquée. S'il est nécessaire de construire un observateur dans l'état *détaché*, le constructeur alternatif peut être utilisé

`Observer.unbound(body: =>Unit): Observer`

Dans ce cas, il sera nécessaire de l'attacher aux signaux qu'il écoute explicitement en invoquant la méthode *bind* sur l'objet retourné.

2.2.10 Contexte de mutation

Un contexte de mutation représente une mutation du graphe de signaux considérée atomique du point de vue des observateurs et des signaux différés. Tant qu'un contexte de mutation est ouvert, aucun observateur ou signal différé n'est calculé. À la place, ceux-ci sont placé en attente jusqu'à la fermeture du contexte de mutation.

Un tel contexte peut être construit explicitement sous la forme

```
1 Signal.atomically {
2   // mutations...
3 }
```

ou implicitement en effectuant la mutation d'un signal Source. Chaque mutation d'une `Source` ouvre en effet un nouveau contexte de mutation puis le ferme immédiatement.

Lorsqu'un contexte de mutation est ouvert tandis qu'un autre contexte est déjà ouvert, le contexte existant est réutilisé. Par exemple lors de la mutation d'une source dans un bloc `Signal.atomically`, le contexte enfant créé par la source sera ignoré et aucun observateur ni signal différé ne sera traité avant la fermeture du contexte extérieur.

L'intérêt de cette construction se révèle en considérant la nature récursive de l'invalidation des signaux. Lorsqu'un signal source est modifié, l'ensemble des ses signaux enfants doit être invalidé. Si l'un de ces signaux utilise la sémantique d'évaluation *différée*, il est nécessaire de le recalculer. S'il possède des observateurs, il est également nécessaire de l'invoquer avant de retourner au code appelant. De plus, chaque signal enfant peut également posséder un ensemble de signaux enfants pour lesquels l'opération doit être répétée. Des structures en diamant sont également possibles, causant une double invalidation d'un signal enfant (figure 2.11).

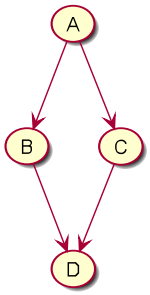

FIGURE 2.11 – Exemple de dépendance en diamant

Dans de telles la situations, la présence du contexte de mutation permet d'as-

surer une sémantique *exactly-once* pour les observateurs et les signaux différés. Un observateur n'est invoqué, ou un signal différé recalculé, qu'une seule et unique fois par contexte de mutation, lorsque celui-ci est fermé.

Plus spécifiquement, les règles suivant s'appliquent :

a Aucun observateur ou signal différé n'est évalué tant qu'un contexte de mutation est ouvert. Il est à la place placé dans une queue interne au contexte.

b Dès lors que le contexte de mutation est fermé, l'ensemble des signaux différés en queue sont évalués et mis à jour. Ces signaux sont évalués dans l'ordre dans lequel ils ont été placés dans la queue, mais un même signal ne sera évalué qu'une seule fois.

c Si un signal en queue dépend de l'état d'un autre signal différé placé après lui dans la queue, cet autre signal est évalué immédiatement et sa position dans la queue est ignorée.

d Une fois l'ensemble des signaux différés évalués, les observateurs sont invoqués dans l'ordre dans lequel ils ont été ajoutés à la queue. De façon similaire aux signaux, un observateur ne sera invoqué qu'une seule et unique fois par contexte.

e La fermeture du contexte est elle-même effectuée dans un nouveau contexte de mutation. Celui-ci est ouvert dès lors que le contexte parent débute sa procédure de fermeture et sera fermé une fois l'ensemble des signaux et observateurs invoqués. S'il n'y a aucun signal ni observateur en queue, un nouveau contexte n'est pas ouvert.

f L'opération de fermeture ne peut pas être interrompue, une exception lancée lors de l'évaluation d'un signal ou d'un observateur interrompt l'opération en cours mais n'affecte pas les autres signaux et observateurs qui seront tout de même.

g Si au moins une exception a été lancée pendant le processus de fermeture d'un contexte de mutation, une exception [7] sera lancée à la fin de l'opération et contiendra l'ensemble des exceptions capturées. Ce mécanisme est conçu pour faciliter le *debug* de l'application et son usage doit être évité. Les exceptions possibles devraient être traitées localement dans les signaux ou les observateurs.

Ces règles ont pour but de simplifier la conceptualisation du comportement d'un graphe de signaux. Chaque mutation s'accompagne au maximum d'une évaluation des observateurs et signaux différés. Cette garantie est particulièrement intéressante lors de l'utilisation de signaux utilisant la sémantique *différée* puisqu'elle permet de rendre des opérations tels que `fold` indépendantes de la structure du graphe, aussi complexe soit-elle.

7. De type `MutationContext.MutationException`

2.2.11 Parallélisme

Le parallélisme est un concept majeur dans le développement logiciel depuis que les processeurs tendent à se paralléliser plus qu'il n'accélèrent. Utiliser efficacement un processeur moderne signifie être capable de répartir les calculs à effectuer sur les multiples cœurs d'exécution qu'il met à disposition.

Avec la parallélisation de l'exécution, des comportements non-déterministes apparaissent. Il n'existe plus aucune garantie concernant l'ordre dans lequel les opérations seront effectuées entre les différents fils d'exécution. Des facteurs externes tel que le système d'exploitation et les autres applications exécutée de façon concurrente par l'utilisateur se partagent tous les ressources du système et affecte l'ordonnancement des opérations.

Il est alors nécessaire d'intégrer à l'application *multi-threadée*, des points de synchronisation ou des verrous pour restaurer un certain nombre de garanties quant à l'ordre d'exécution des instructions. Ces mécanismes sont généralement coûteux en terme de performances et leur usage excessif peut conduire à une exécution effectivement plus lente que lors de l'utilisation d'un unique *thread*. L'*overhead* de la synchronisation éclipsant alors le gain de performance apporté par le parallélisme.

La question se pose alors de l'utilisation des signaux dans un contexte parallèle. Dans le cadre de ce travail, les signaux ont été utilisés pour la construction de l'interface utilisateur côté client. JavaScript est un environnement *single threaded* n'offrant aucun mécanisme de parallélisation [8]. La concurrence n'était donc pas un problème.

Cependant, les signaux ne sont pas limités à la construction d'interfaces web et peuvent être utilisés pour modéliser des dépendances complexes dans un ensemble d'informations manipulées par une application possiblement parallèle. Afin d'étudier les implications de la parallélisation, deux cas d'utilisation ont été imaginés :

1. L'utilisation *multi-threadée* d'un unique graphe de signaux
2. L'utilisation des signaux comme mécanisme de parallélisation

Le sections ci-après présentent une analyse théorique de ces deux approches avec des cas d'usages potentiels.

8. Plus spécifiquement, il n'existe pas de parallélisation avec mémoire partagée. L'environnement web défini les *Web Workers* qui sont effectivement des processus concurrents, mais dont la communication avec le processus principal se fait par échange de messages et événements.

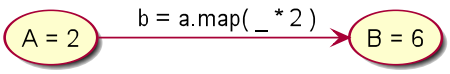

FIGURE 2.12 – Signaux incohérents

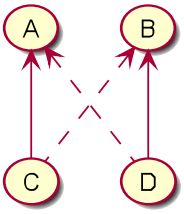

FIGURE 2.13 – Situation d'inter-blocage potentielle

Utilisation *multi-threadée* d'un unique graphe de signaux

Dans cette situation, un graphe unique de signaux est accédé parallèlement par de multiples *threads* de l'application. La tâche de parallélisation est attribuée à l'utilisateur de la bibliothèque qui est alors responsable de déterminer le nombre de *threads* à exécuter et leurs tâches respectives. Il attend du graphe de signaux un certain nombre de garanties :

— Cohérence : les données stockées dans chaque nœud du graphe doivent rester cohérentes par rapport à l'ensemble du graphe. Si un signal dépend d'un autre et que l'état de ces deux signaux est actuellement en cache, les valeurs de ces deux signaux doit correspondra à la relation qui les unis. La figure 2.12 illustre une telle situation où les données en cache ne correspondent pas à la relation.

  Cette incohérence peut provenir d'un croisement des opérations d'invalidations et de calcul d'un signal *lazy*. Si un signal d'invalidation est reçu par un nœud tandis que qu'un autre *thread* est déjà en train de recalculer ce même nœud, il est possible que l'invalidation soit ignorée, malgré que le calcul soit effectué avec des données périmées.

— Absence d'inter-blocage : une solution pour résoudre les problèmes de cohérence est l'introduction de synchronisation et de verrous. Il est cependant important de ne pas verrouiller *trop*. Si chaque accès aux signaux nécessite le verrouillage de l'ensemble du graphe, les différents fils d'exécutions se retrouveraient en attente permanente et les avantages du parallélisme seraient perdus.

  À l'inverse, dans le cas d'un verrouillage fin, nœud par nœud, il est alors important de s'assurer de l'absence d'inter-blocage, les mises à jour d'un graphe de signal pouvant être initiées parallèlement à partir de multiples nœuds.

La figure 2.13 illustre une telle situation. Si le signal c, en étant calculé, verrouille le signal a tandis que le signal d verrouille le signal b de façon concurrente, les deux signaux se retrouvent bloqués en attente du second signal nécessaire à leur évaluation.

Une solution potentielle à cette situation est d'imposer un ordre dans l'acquisition des verrous entre les signaux, par exemple sur la base du `hashCode` de l'objet signal étant verrouillé. La question de savoir si cela est suffisant pour garantir l'absence d'inter-blocage dans le graphe n'a pas été approfondie.

— Sémantique des observateurs : dans une situation non-parallèle, un observateur est invoqué *exactement une fois* lors de la fermeture du contexte de mutation et avant le retour au code utilisateur ayant initié la mutation. Que faire lorsque de multiples nœuds sont mis à jour de façon parallèle avec des observateurs attachés ?

Faut-il exécuter l'observateur une fois par mutation, avec des données potentiellement périmées ? Faut-il plutôt différer l'exécution de l'observateur jusqu'à ce que l'ensemble du graphe soit stabilisé ? Dans ce cas, quel *thread* est responsable d'invoquer l'observateur ? Que faire si le graphe ne se stabilise jamais, c'est-à-dire que des modifications sont continuellement apportées aux données, l'observateur se retrouverait alors différé indéfiniment ?

La problématique est identique en ce qui concerne les signaux différés puisqu'ils partagent leur sémantique d'évaluation avec les observateurs. Est-ce que cela a toujours un sens dans un environnement parallèle ou est-il nécessaire d'établir des règles supplémentaires, spécifiques aux signaux différés ?

Toutes ces questions attestent de la difficulté d'établir des règles d'évaluation claires et simples dans un contexte *multi-threadé*. Ce qui correspond à un cas d'utilisation peut ne pas être adapté à un autre.

Les approches les plus simples, comme par exemple exécuter les observateurs une fois par mutation sur le *thread* ayant initié la mutation et sans se préoccuper d'exposer des données périmées dans le graphe, sont les plus simples à implémenter mais également les moins utiles pour le développeur. Les approches plus strictes, plus utiles au développeur, sont également plus complexes à implémenter. L'impact sur les performances est également à considérer.

Utilisation des signaux comme mécanisme de parallélisation

Une approche alternative de la question du parallélisme consiste à se servir du graphe de signaux comme primitive de parallélisation. Dans ce cas, ce n'est plus le développeur qui est responsable de paralléliser son application,

mais la bibliothèque de signaux. L'objectif est alors de calculer automatiquement les différents nœuds du graphe de façon concurrente.

Une approche possible serait de modifier la signature des méthodes d'accès à l'état d'un signal pour retourner des instances de `Future`. En utilisant le mécanisme d'`ExecutionContext` de la bibliothèque Scala, la gestion du *thread pool* est déléguée à la plateforme, tout en laissant au développeur la possibilité de configurer son fonctionnement si nécessaire.

En ce qui concerne les observateurs, une approche intéressante serait d'associer à chaque mutation une invocation des observateurs, au cours de laquelle le graphe est dans un état cohérent à la suite de cette mutation. L'objectif est ici de pouvoir utiliser un graphe de signaux comme implémentation d'une fonction de la forme `(A, B, C, ...) => Future[Z]`.

Une telle fonction serait construite autour d'un graphe disposant d'une source par paramètre et d'un signal feuille exprimant la valeur de retour par transformation des signaux sources, potentiellement avec des nombreux niveaux de signaux intermédiaires. L'ensemble des sources seraient alors mises à jour de façon atomique, provoquant une mise à jour de l'ensembles des signaux du graphe puis une invocation d'un observateur attaché au nœud feuille. La figure 2.14 présente un exemple de ce concept.

```
1  val A = Source.undefined[String]
2  val B = Source.undefined[Int]
3  // ... signaux intermédiaires
4  val Z = Signal { ... }
5
6  def foo(a: String, b: Int): Future[Boolean] = {
7     val p = new Promise[Boolean]
8     Signal.atomically {
9        A := a
10       B := b
11       Observer.once(Z) { res => p.resolve(res) }
12    }
13    p.future
14 }
```

FIGURE 2.14 – Exemple d'utilisation d'un graphe de signaux comme mécanisme de parallélisation

Les deux approches peuvent également être combinées. Si l'observateur invoqué sur le nœud feuille observe un état cohérent du graphe même dans le cas d'invocations concurrentes, le graphe fonctionne alors comme un *pipeline* d'évaluation où les données sont propagées le long des signaux jusqu'à attendre le nœud feuille avec la possibilité de débuter un nouveau calcul

au niveau des racines tandis que le premier n'a pas encore atteint l'extrémité du graphe. D'une façon, dès lors que tous les enfants d'un signal ont été calculés, il est possible de débuter un nouveau calcul au niveau de ce signal.

Par exemple, dans la structure en diamant de la figure 2.11, le signal D dépend indirectement de la valeur de A. Cependant, une fois que les valeurs de B et C ont été calculées, le signal A peut être réutilisé pour le prochain calcul sans affecter le calcul de D puisque la valeur correcte, de son point de vue, de A a été *capturée* dans les valeur de B et C. Une fois le calcul de D terminé, les signaux B et C peuvent à leur tour être réutilisés pour le prochain calcul, sans attendre la fin de la propagation.

2.3 Solutions existantes

2.3.1 ReactiveX

Dans les implémentations les plus populaires de framework réactifs-fonctionnels, dont notamment la bibliothèque *ReactiveX* disponible pour de nombreux langages, on retrouve généralement un concept similaire sous le nom de `Rx`.

Bien que tous deux soient une abstraction du temps, les `Rx` représentent fondamentalement une séquence tandis qu'un `Signal` est une valeur unique à un instant donné. Afin de clarifier cette distinction, ces deux interfaces peuvent être comparées à leur équivalent non-réactif le plus proche dans la bibliothèque standard de Scala :

Type de base	Type réactif	Différence
`Stream[T]`	`Rx[T]`	Tous les éléments d'un `Stream` sont calculables à l'instant présent, les éléments d'une `Rx` ne sont peut être pas encore disponibles
`Future[T]`	`Signal[T]`	Un `Future` n'est résolu qu'une seule fois, un `Signal` peut changer de valeur de multiples fois au cours du temps

Ainsi, certaines opérations monadiques, dont la sémantique peut être ambiguë dans le cas des valeurs réactives, peuvent présenter des comportements considérablement différents entre leur implémentation par les signaux et les

`Rx`s. C'est le cas notamment de la fonction `flatMap`[9] :

— Sur une valeur `Rx`, l'opération `flatMap` effectue une concaténation des valeurs des sous-séquences retournées, potentiellement entrelacées.

— Sur un `Signal`, l'opération `flatMap` se comporte comme un switch : le signal original est utilisé pour sélectionner un second signal, dont la valeur sera celle du signal produit par `flatMap`. Lorsque la valeur du signal initial change, la sélection est effectuée à nouveau et le signal résultant est utilisé comme nouvelle source de valeurs.

Bien que la bibliothèque *ReactiveX* propose un opérateur spécifique avec la sémantique de switch, il est utile de préciser que l'opération `flatMap` est l'opération utilisée par le langage Scala lors de l'utilisation de multiples générateurs dans une compréhension `for`.

Ainsi le code

```
1 val a: Signal[Int] = ...
2 val b: Array[Signal[Double]] = ...
3 val c: Signal[Double] = for (x <- a; y <- b(x)) yield y * 2
```

sera transformé par le compilateur en

```
1 val c = a.flatMap(x => b(x)).map(y * 2)
```

et produira des résultats radicalement différents selon l'abstraction utilisée :

— Dans le cas des `Rx`, le résultat est l'agglomération des tous les flux de valeurs de `b` sélectionnés au fil du temps, potentiellement de multiples fois. Le résultat n'est très probablement pas celui escompté.

— Dans le cas de `Signal`, le résultat est un signal `c` dont la valeur est le double de celle d'un signal contenu dans le tableau `b` et désigné par la valeur du signal `a`, le tout sur la base des valeurs de ces signaux au moment présent. Dans le futur, lorsque la valeur de `a` ou de `b(x)` changera, la valeur de `c` sera également mise à jour.

La sémantique des signaux, basée sur les valeurs plutôt que la séquence de ces valeurs, a pour but d'être la plus adaptée possible à l'usage qui en est fait dans ce travail, c'est à dire la conception d'interfaces graphique.

Une conséquence supplémentaire de cette différence entre signaux et `Rx` est la possibilité de définir aisément le concept de *signal expression* dans le cas des

9. `Signal[T].flatMap[U](fn: T => U): Signal[U]`, de façon similaire pour `Rx[T]`

signaux. Dans le cas des `Rx`, le comportement attendu d'une telle construction n'est pas évident dans une situation où les flux sont potentiellement finis.

2.3.2 Scala.rx

Scala.rx est l'implémentation la plus proche des signaux utilisés dans ce projet. La simplicité d'utilisation est particulièrement appréciable avec la méthode de définition de `Rx`s sur la base d'expressions arbitraires.

Les différences sont principalement liées aux limitations volontairement imposées dans Scala.rx.

Pas de variable réactive non-définie / vide

La raison invoquée est la difficulté d'intégrer l'absence de valeur de façon intuitive dans la syntaxe déclarative. Plus spécifiquement, l'auteur mentionne plusieurs solutions potentielles :

1. Bloquer le thread courant jusqu'à ce qu'une valeur soit disponible. Cette solution n'est cependant pas envisageable pour une implémentation visant l'environnement Javascript.
2. Lancer une exception lors de l'accès à une variable vide, mais recommencer l'opération une fois que sa valeur sera définie. Cette option présente le désavantage que le calcul d'une variable réactive puisse potentiellement être démarré puis interrompu de multiples fois si de nombreuses dépendances sont indéfinies.
3. Obliger le style monadique et retirer la méthode implicite de définition. Cette option n'est pas considérée pour des raisons d'expérience utilisateur évidentes.
4. Utiliser un plugin du compilateur pour transformer automatiquement le code en style monadique. Cette approche présente cependant de nombreux défis techniques et requiert une étape supplémentaire de configuration indésirable.

En pratique l'absence de variable réactive vide est une gêne considérable dans le domaine du web où toutes les opérations sont asynchrones et non-bloquante. Le concept de promesse est omni-présent et il est ainsi difficile d'associer les interfaces natives du navigateur avec le réseau de variables réactives.

Une première approche est d'utiliser directement le type `Option` de Scala pour exprimer l'absence d'information le temps de l'opération. Mais cette idée impose une verbosité beaucoup plus importante du code puisqu'il est

maintenant nécessaire de manipuler explicitement des `Rx[Option[T]]` au lieu de simples `Rx[T]` et de laisser le soin de la gestion de l'asynchrone à la bibliothèque.

De plus, la seconde approche proposée (interrompre le calcul par une exception) ne semble pas déraisonnable et peut constituer une approche valide si correctement documentée. Il est important que les conséquences d'une variable indéfinie soient connues et considérées, mais leur absence volontaire semble au final plus gênante que bénéfique. Par ailleurs, le style monadique ne présente pas les inconvénients mentionnés et constitue une alternative valide si le style est plus adapté à un problème spécifique.

Complexité de l'implémentation

Malgré l'effort fourni pour proposer une interface simple et efficace, l'implémentation est en réalité relativement complexe et très largement basée sur l'utilisation de macros. Le code écrit par le développeur n'est pas le code finalement exécuté. Une raison probable de cette implémentation est basée sur la méthode choisie pour détecter automatiquement les dépendances entre variables réactives avec le style déclaratif.

Dans Scala.rx, les expressions qui définissent des variables réactives sont réécrite sous formes de fonctions exploitant largement le système de paramètres implicites pour transmettre le contexte d'accès à une variable réactive entre parent et enfant. Bien que cela soit une approche très proche des standards en Scala, elle requiert une couche syntaxique supplémentaire ou l'utilisation de macros afin de l'ajouter automatiquement. Le système résultant est excessivement complexe et les erreurs rencontrées lors de l'usage ne sont pas intuitives puisqu'elles ne correspondent pas au code écrit.

De plus, l'approche n'est pas portable entre les versions de Scala puisqu'elle dépend de l'implémentation interne du compilateur. Dans le cas de Scala.rx, l'implémentation est compilée simultanément pour les version 2.10, 2.11 et 2.12 de Scala et requiert pour chaque version une implémentation différente des macros pour s'aligner avec les changements apportés au compilateur et à la syntaxe du langage.

Une approche alternative est possible : `DynamicVariable`. Cette classe de la bibliothèque standard de Scala est rarement utilisée puisque très spécifique et généralement moins explicite que l'usage de paramètres implicites qui sont plus en accord avec les principes de la programmation fonctionnelle. Elle remplit cependant un rôle semblable en offrant une sémantique de variable à portée dynamique plutôt que lexicale. Ce mécanisme permet de déplacer la transmission du contexte d'évaluation des paramètres implicites à un canal annexe dédié.

Le code est ainsi fortement simplifié puisque la transmission du contexte est clairement dissociée, évitant la nécessité de modifier le code utilisateur et donc l'usage de macros au prix d'une architecture pouvant être considérée moins pure d'un point de vue fonctionnel. L'absence de macros permet également une compatibilité plus aisée avec les futures versions du langage.

En contrepartie, la détection automatique des dépendances dépend de l'exécution synchrone de l'expression de définition (§ 2.2.3). De façon générale, effectuer le calcul de la valeur d'un signal de façon différée par rapport à l'instant d'évaluation choisi par la bibliothèque ne semble pas compatible avec les objectifs de fournir un modèle d'évaluation strict et prévisible.

2.4 Implémentation

2.4.1 Hiérarchie complète

La figure 2.4.1 présente la hiérarchie formée par les différentes classes de signaux. Cette section aborde plus en détails les spécificités de chacune d'elles.

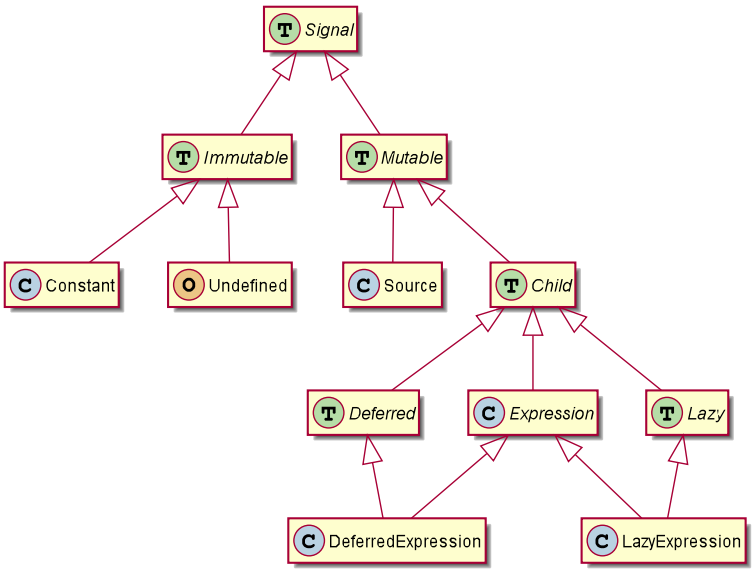

FIGURE 2.15 – Hiérarchie des signaux

L'implémentation des signaux est organisée sur la base du raffinement successif de la sémantique d'un signal. Chaque niveau laisse généralement un point de sa sémantique indéfini, sous la forme d'une méthode abstraite à implémenter par les sous-classes.

— `Signal` : la vue la plus générale d'un signal, ce trait définit les méthodes de transformation et d'accès à l'état du signal. La fonction `def option: T` est indéfinie et doit retourner l'état courante du signal.

— `Immutable` : redéfinit `option` en `val` puisque l'état est immutable, mais laisse la valeur abstraite.

— `Constant` : définit `val option = Some(value)`

— `Undefined` : définit `val option = None`

— `Mutable` : déclare la méthode abstraite `def current: Option[T]`, représentant également l'état du signal mais dont le nom indique clairement la mutabilité du signal.

  Définit la méthode `option` en se basant sur `current` mais y ajoute la détection automatique des signaux enfants. De cette façon, toutes les méthodes de `Signal`, en utilisant directement ou indirectement `option`, obtiennent également les mécanismes de détection.

— `Source` : définit une variable interne pour stocker l'état de la source, implémente `current` sur la base de cette variable.

  Ajoute également les opérations de mutation explicites.

— `Child` : déclare la méthode abstraite `def generate: Option[T]`, appelée lorsque l'état du signal doit être calculé. Implémente `current` sur la base de `generate` et y ajoute le mécanisme de *memoization*.

  Définit la méthode `def invalidate(): Unit` qui permet d'effacer l'état mémorisé du signal et d'invalider récursivement tous les signaux enfants. Cette méthode est en principe appelée par `Mutable`.

— `Lazy` : trait *marqueur*, l'implémentation de `Child` utilise déjà la sémantique *lazy* par défaut. Il n'y a donc rien à modifier.

— `Deferred` : surcharge la méthode `invalidate` définie par `Child` en insérant le signal dans la queue du contexte de mutation courant. De cette façon, ce signal sera recalculé de façon différée à la fermeture du contexte de mutation.

— `Expression` : implémente `generate` sur la base de l'expression passée en paramètre.

— `LazyExpr` : combine `Expression` avec `Lazy`

— `DeferredExpr` : combine `Expression` avec `Deferred`

Les classes `LazyExpr` et `DeferredExpr` sont privées. S'il est nécessaire dans le code utilisateur de déterminer la sémantique d'une expression, un *pattern matching* sur les traits `Lazy` et `Deferred` peut être effectué. Il est également possible de tester si un signal est une expression constante avec l'opérateur `with`, sous la forme : `case e: Expression with Lazy`.

2.4.2 Détection automatique des dépendances

La détection automatique des dépendances des signaux et observateurs joue un rôle essentiel dans l'implémentation des signaux. Même à l'intérieur de la bibliothèque, aucune dépendance n'est déclarée explicitement et le détection automatique est utilisée.

L'élément clé de ce mécanisme est la classe `x.s.t.TracingContext`. Un contexte de traçage est un mécanisme très similaire à celui du contexte de mutation (§ 2.2.10). Les deux sont construits sur la base du mécanisme de `DynamicVariable` présent dans la bibliothèque standard Scala.

Une opération de traçage est effectuée par un appel à la méthode

```
def TracingContext.trace[T](expr: =>T): (T, List[Mutable[_]])
```

prenant en paramètre une expression arbitraire qui sera évaluée. La méthode retourne un couple de valeurs, dont le premier membre est le résultat de l'évaluation de l'expression et le second la liste de tous les signaux qui ont été accédés lors de cette évaluation.

Il convient de noter que seuls des signaux de type `Mutable[_]` sont retournés. En effet, cette liste à pour but d'énumérer les dépendances de l'expression évaluée et ainsi permettre la mise en place des notifications de mises à jour. Un signal immutable ne changeant par définition jamais, il n'y a pas d'intérêt à le considérer comme une dépendance de l'expression.

La classe `DynamicVariable` est conçue pour représenter une variable dont la valeur et déterminée par portée dynamique plutôt que portée lexicale. C'est-à-dire que la valeur de la variable est déterminée dynamiquement selon la pile d'appel à l'exécution plutôt que par la structure du code source.

En pratique, il n'existe pas de portée dynamique en Scala, le concept est donc émulé en utilisant une pile locale au thread courant afin de stocker la valeur de la variable. La méthode

```
def DynamicVariable[T].withValue[U](v: T)(expr: =>U): U
```

permet l'exécution d'un bloc de code arbitraire durant laquelle la méthode `DynamicVariable.value` retournera la valeur passée en paramètre. Chaque invocation de la méthode `withValue` ajoute un nouvel étage à la pile qui sera dépilé une fois l'évaluation terminée.

Dans le cas de `TracingContext`, un nouveau contexte est créé lors de l'appel à la méthode `trace` et placé dans une variable `DynamicVariable`. La seconde partie du système est implémentée dans la classe `Mutable`. Comme indiqué dans le détail de la hiérarchie des signaux (§ 2.4.1), cette classe implémente

la méthode *option* de `Signal` en y ajoutant le mécanisme de détection des dépendances.

Plus spécifiquement, chaque invocation de la méthode `option` d'un signal `Mutable` va appeler la méthode interne `TracingContext.record` afin d'enregistrer l'accès au signal dans le contexte courant. Déterminer le *contexte courant* consiste en un simple accès à la variable `DynamicVariable` mentionnée précédemment.

Contrairement aux contextes de mutations, l'imbrication des contextes de traçage est significative. L'accès à un signal ne sera en effet enregistré que dans le contexte le plus récent, au sommet de la pile. Ceci découle de l'utilisation de contextes de traçage, en interne, par les signaux expression.

L'objectif final est de limiter la liste des signaux retournés par la méthode `trace` aux dépendances directes de l'expression évaluée. Il n'est en effet pas intéressant de récupérer les dépendances *transitives* d'un signal puisque les notification de mises à jour sont propagées récursivement jusqu'aux feuilles du graphe.

La figure 2.16 illustre une situation d'imbrication des contextes de traçage. L'évaluation du signal E crée un nouveau contexte qui sera utilisé pour déterminer ses dépendances. Lors de l'évaluation du signal C, un nouveau contexte est créé pour déterminer les dépendances de ce signal. Les signaux B et A forment ainsi des dépendances *transitives* de E et ne seront pas retournées dans la liste de signaux accédés par le signal E, celle-ci se limitant aux signaux D et C.

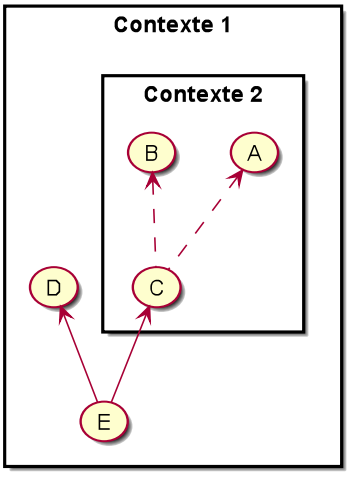

FIGURE 2.16 – Imbrication des contextes de traçage

Cette approche est une alternative à l'utilisation de paramètres implicites,

tels qu'utilisés par la bibliothèque *Scala.rx* [6]. Les variables dynamiques forment un canal indépendant pour déterminer les relations de dépendance entre signaux et observateurs. Il n'est alors plus nécessaire de recourir à des macros compilateur pour transformer automatiquement l'expression fournie par l'utilisateur en fonction anonyme.

En contre-partie, la détection de dépendances est maintenant liée à la structure de la pile d'appels lors de l'accès aux signaux, ce qui implique une exécution synchrone des opérations. Les structures *lazy* ou *asynchrones* sont donc particulièrement problématiques puisqu'elles ne conservent pas la variable dynamique qui sera perdue dès la retour de l'appel synchrone. Dans un environnement *single-threaded* tel que JavaScript, ceci n'est cependant pas un problème.

2.4.3 Implémentation des opérateurs de transformation

Lors de l'implémentation des opérateurs de transformation, deux approches ont été envisagées :

1. Implémentation d'une sous-classe de `Signal` par transformation, par exemple des classes `Map`, `Fold`, etc. Cette approche est par exemple utilisée par la bibliothèque *Bindings.scala* [27].
2. Réutiliser le mécanisme des signaux expressions.

La seconde approche a été choisie sur la base de la réutilisation de l'ensemble des mécanismes développés pour les signaux expression. Le seul élément à considérer est alors l'implémentation concrète de la transformation. Les détails annexes, tels que la sémantique d'évaluation ou la gestion des dépendances est obtenue de façon automatique.

Par exemple, la transformation `map` est déclarée simplement par

```
def map[U](f: T =>U): Signal[U] =
  Signal.define(option.map(f))
```

Ici, `option` fait référence à la méthode `Signal.option`. Le type de signal retourné par une transformation dépend du type de signal sur lequel elle est invoquée. Appliquer une transformation `map` sur un signal `Constant` retourne un nouveau signal `Constant`. Ceci résulte du fait que, lors du calcul de l'expression de définition de la transformation, aucun signal `Mutable` n'a été accédé. Le constructeur `Signal.define` retourne alors un signal de type `Constant` ce qui correspond au comportement attendu de la transformation.

Dans une version antérieure de l'implémentation des signaux, ces transformations étaient déclarées de façon abstraite dans l'interface `Signal` puis implémentées indépendamment dans les sous-classes `Mutable` et `Immutable`.

```
val a: Constant[Int] = Constant(2)
val b: Signal[Int] = ...

val c = a.map(_ * 2)
// `c` est un signal `Constant[Int]`

val d = a.map(_ * b.value)
// `d` est un signal `Expression[Int]`
```

FIGURE 2.17 – Exemple de transformation d'un signal constant en utilisant des fonctions pures ou impures

Avec l'amélioration du constructeur `Signal.define`, prenant en compte les signaux accédés lors de la définition d'un nouveau signal, cette complexité a pu être éliminée et l'ensemble des signaux, mutables comme immutables, partagent maintenant une définition unique dans la classe parente `Signal`.

Un effet secondaire supplémentaire de cette implémentation est le support de fonctions de transformation impures. Référencer la valeur d'autres signaux dans une fonction passée aux opérateurs produira le résultat attendu. La figure 2.17 illustre une telle situation où un même signal `Constant` est transformé en un nouveau signal `Constant` ou une `Expression` selon la pureté de la fonction passée en paramètre. Le support de ce style, bien que d'une élégance discutable, permet de réduire le risque de surprendre un utilisateur qui s'attendrait à pouvoir se servir de telles constructions.

2.4.4 Modèle push, pull ou hybride

Deux modes de fonctionnement sont généralement décrits pour des systèmes fonctionnels-réactifs : *push* et *pull*.

L'approche *push* se base sur les changements apportés aux signaux sources pour recalculer tous les signaux enfants qui en dépendent. Dans l'approche *pull*, c'est l'accès aux signaux enfants qui provoque le calcul des valeurs intermédiaires jusqu'aux signaux sources. Dans les deux cas, des opérations potentiellement inutiles ou redondantes sont effectuées.

L'approche mixte *push-pull* se base sur une approche principalement *pull* où l'accès à l'état d'un signal déclenche son évaluation, à laquelle vient s'ajouter un mécanisme de *memoization* qui maintient l'état courant du signal après son calcul. L'invalidation de ces caches se fait ensuite selon une approche *push* : un changement d'état des signaux sources est notifié à toutes les dépendances de façon récursive.

Dans le cadre de ce projet, l'approche *pull* était tout simplement inutilisable. Lors de l'implémentation d'une interface utilisateur, les notifications de mises à jour proviennent des racines du graphe de signaux. L'interface ne connaît *a priori* pas les moments opportuns à l'exécution d'un *pull* pour se mettre à jour. L'approche *push* est déjà plus adaptée, mais impose une mise à jour complète du graphe à chaque changement.

L'architecture proposée de l'application associe un graphe de signaux, représentant l'ensemble des données disponibles, avec un ensemble d'observateurs, correspondant aux données actuellement utilisées par l'interface. Il est courant qu'une partie de ce graphe soit inutilisée dans l'état courant de l'interface. Lorsque l'utilisateur navigue dans l'application, les observateurs des données qui ne sont plus visibles sont retirés tandis que de nouveaux sont attachés sur les données qui apparaissent à l'écran. Ainsi, les sections du graphe considérées actives changent dynamiquement en fonction des actions effectuées par l'utilisateur.

Le modèle hybride permet aux sections actives du graphe, celles dont les feuilles disposent d'observateurs, de se comporter effectivement de façon *push* en étant recalculées dès lors que les données sources sont modifiées. À l'inverse, les sections inactives du graphe se comportent de façon *pull*, n'étant pas recalculées inutilement tant qu'un observateur n'y aura pas été attaché ou leur état courant accédé de façon explicite.

Chapitre 3

Framework web

3.1 Motivations

La construction de l'interface utilisateur d'un logiciel est une tâche essentielle mais néanmoins fastidieuse. Les technologies classiques telles que *Swing* sur la plateforme Java construisent ces interfaces de façon impérative, en utilisant le même langage que celui utilisé pour le développement de la partie métier de l'application. Cependant, un langage adapté à la définition d'algorithmes traitant des données ne l'est pas forcément à la définition d'interfaces graphiques pour l'utilisateur. Le code résultant est alors excessivement long, verbeux et répétitif.

La tendance actuelle tend à dissocier logique métier et interface en offrant au développeur un langage alternatif, adapté à la construction d'interface de façon déclarative, de façon similaire à la construction d'une page web en HTML. C'est par exemple le cas de *JavaFX* utilisant un document XML pour décrire la structure de l'interface et le langage CSS pour définir son apparence.

Des mécanismes de *data-binding* sont ensuite fournis pour injecter les données produites par l'application à des emplacements spécifiques de l'interface. Le framework s'occupant ensuite de la mise à jour automatique de l'interface lorsque les données source changent [1].

Dans le monde du web, malgré l'usage universel des langages HTML et CSS, aucun mécanisme officiel de *data-binding* n'est disponible. Il est donc de la responsabilité du développeur de manipuler explicitement la structure du document avec l'interface du DOM pour le maintenir à jour avec ses données métier, ou de réutiliser une bibliothèque existante.

1. En *JavaFX*, l'interface `Observable` est à la base de ce mécanisme.

L'introduction des technologies *Web Components* étend les possibilités de la plateforme et rend le développement de composants réutilisables considérablement plus aisé, mais n'adresse pas la question du *data-binding*. Le développeur est toujours responsable de cet aspect pourtant complexe de son application.

La deuxième partie de ce travail consistait donc à développer une bibliothèque Scala.js permettant la construction de composants conformes aux standards *Web Components*, mais sous une forme adaptée au langage Scala. Les langages HTML et CSS sont utilisés pour décrire la structure et l'apparence du composant plutôt que de tenter de réutiliser le langage Scala, comme d'autre projet ont pu le faire [2].

La syntaxe HTML utilisée dans la définition des templates est étendue pour y intégrer des mécanismes de *data-binding* implémentés par la bibliothèque. Cette approche est similaire à la philosophie adoptée par le framework *Angular* [26] : le langage HTML est à la base du web, pourquoi en inventer un nouveau plutôt que d'étendre ce qui est déjà universellement utilisé.

Finalement, l'implémentation des signaux développées précédemment est réutilisée comme mécanisme de propagation des changements entre le code métier et l'interface. La bibliothèque se charge se configurer les observateurs nécessaires en fonction des données injectées dans l'interface et s'occupe des mises à jour nécessaires lorsque ces données changent.

3.2 À propos des standards Web Components

L'implémentation de la partie web de Xuen dépend extensivement des mécanismes introduits par les spécifications *Shadow DOM* [12], *Custom Elements* [14] et les spécifications annexes telles que *CSS Scoping* [13].

Ces standards définissent des mécanismes permettant la définition de nouveaux éléments personnalisés, encapsulés et réutilisables. Ils définissent également comment l'encapsulation du composant est implémentée au niveau des styles CSS et des événements DOM. Ces mécanismes sont réutilisés avec le minimum d'abstraction lorsque cela est possible pour Xuen.

Définir un élément Xuen (§ 3.3.2) correspond à la définition d'un nouvel élément *Custom Elements* : la classe produite est passée à la méthode native `define()` du navigateur tel que le serait une classe construite directement en JavaScript.

Les templates se basent sur *Shadow DOM* comme mécanisme d'encapsulation. Ils peuvent ainsi réutiliser le concept de `<slot>` introduit par ce

2. Par exemple en utilisant des bibliothèques telles que *ScalaTags* [24] et *ScalaCSS* [25]

standard comme primitive de composition. L'encapsulation standard des événements et des styles s'applique également.

Ces concepts sont répartis entre une multitude de spécifications, provenant de différentes organisations et peuvent être difficiles à aborder de prime abord. Néanmoins, une connaissance des ces mécanismes peut se révéler très utile à la compréhension de l'implémentation et du fonctionnement de Xuen.

En guise d'introduction, le guide *Shadow DOM v1 : Self-Contained Web Components* [15] rédigé par Google dans la série *Web Fundamentals* est un bon tour d'horizon des mécanismes liés à Shadow DOM.

3.3 Spécifications

Notation abrégée des packages

Dans la suite de cette section, afin de réduire la longueur des noms des packages, une notation abrégée est utilisée à la place du nom complet d'une classe :

- Le prefix `xuen` est abrégé en `x`.
- De façon similaire, le second niveau est abrégé en une seule lettre.
  - `component` devient `c`,
  - `expression` devient `e`,
  - `template` devient `t`,
  - etc.

Ainsi, par exemple, la classe `xuen.component.Element` est référencée par `x.c.Element`.

3.3.1 Composant

Un composant est la brique essentielle de construction d'une application. Il correspond directement à une définition d'un *Custom Element* tel que défini par les standards web. Une fois défini, un composant peut être instancié un nombre quelconque de fois, selon les besoins de l'application.

Un composant est défini à partir de :

- Un sélecteur : correspondant à la balise HTML qui sera définie pour ce composant. Ce nom doit comporter un tiret (selon la spécification *Custom Elements*).

— Une implémentation (§ 3.3.2) : une sous-classe de `x.c.Element`, définissant le comportement des instances de ce composant.

Optionnellement, un composant peut également posséder :

— Un template (§ 3.3.3) : une structure HTML pouvant contenir des expressions de *data-binding* avec des données réactives sous la forme de signaux, ce template sera matérialisé pour chaque instance du composant.

— Une feuille de styles (§ 3.3.5) : pouvant être utilisée pour définir l'apparence visuelle du composant.

— Une liste de dépendances : une liste d'autres composants devant être chargés avant ce composant, cette liste correspond à d'autres composants utilisés dans le template de ce composant.

Déclaration

En pratique, un composant est créé en déclarant un `object` qui étend la classe `x.c.Component`. La classe correspondant à l'implémentation du composant est généralement déclarée simultanément avec le même nom, formant ainsi une paire (classe, objet compagnon) fréquente en Scala.

```
1  import xuen.component._
2
3  class HelloWorld extends Element(HelloWorld)
4
5  object HelloWorld extends Component[HelloWorld](
6    selector = "hello-world",
7    template = html"""
8      Hello, world!
9    """,
10   stylesheet = css"""
11     :host { color: blue; }
12   """
13 )
```

Ce court exemple illustre déjà la plupart des concepts utilisés dans la construction de composants Xuen.

Interpolateurs `html` et `css`

Lors de la définition du template et de la feuille de style d'un composant, les interpolateurs `html` et `css` sont généralement utilisés. Ils sont une façon simple d'obtenir les objets de type `Template` et `Stylesheet` attendus par le constructeur de `Component` à partir de code source HTML ou CSS.

Ces interpolateurs sont définis par la classe `x.c.Interpolations`. En Scala, l'implémentation de tels interpolateurs repose sur l'utilisation de conversions implicites qui ne sont généralement pas identifiées automatiquement par l'IDE. Il est ainsi nécessaire d'importer cette classe manuellement afin de les utiliser. Alternativement, il est possible d'importer l'ensemble du package `xuen.component` en utilisant une importation *wildcard*.

```
1 import xuen.component._
```

Cette méthode est généralement préférée à l'importation explicite.

Le nom d'interpolateur est trompeur : contrairement à l'interpolateur `s` de la bibliothèque Scala, `html` et `css` ne supportent pas l'insertion de fragments à l'aide du symbole spécial `$`. Ils sont cependant une façon concise d'appliquer un traitement à une chaîne de caractères, en l'occurrence la transformation en instance de `Template` ou `Stylesheet`. Ils sont également l'occasion pour l'IDE d'identifier le langage utilisé dans la chaîne de caractères et ainsi offrir une coloration syntaxique et une auto-complétion appropriée.

Dans le cas d'IntelliJ IDEA, il est possible d'associer des langages arbitraires avec un interpolateur. Il est ainsi possible d'associer l'interpolateur `html` avec le langage HTML, de même pour `css` avec CSS. Dès lors, le code du template ou de la feuille de style sera correctement traité comme HTML ou CSS dans l'éditeur.

Enregistrement et instanciation

L'enregistrement du composant en tant que *custom element* au niveau du navigateur se fait automatiquement lors de la construction de l'objet singleton `x.c.Component` de ce composant.

En Scala, la construction d'un `object` est différée jusqu'à la première référence de cet élément dans le code source, de façon similaire à l'initialisation d'une `lazy val`. La simple présence de la définition dans le code source n'est donc pas suffisante pour que ce composant soit enregistré. La méthode par laquelle le composant est instancié est alors importante.

Un composant peut être instancié de 4 façons différentes :

1. Par l'utilisation du constructeur de son implémentation :

```
1 val element = new HelloWorld
```

2. En utilisant la méthode `instantiate` de `Component` :

```
1 val element = HelloWorld.instantiate()
```

3. En utilisant la méthode `createElement` de `Document` :

```
val element = dom.document.createElement("hello-world")
```

4. Implicitement par le parser HTML :

```
val div = dom.document.createElement("div")
div.innerHTML = "<hello-world></hello-world>"
```

Dans les deux premiers cas, l'objet `Component` est référencé directement ou indirectement et il n'est pas nécessaire de se préoccuper de l'enregistrement. Ce sont les méthodes préférées lorsque le composant est instancié par le code du développeur et non le navigateur lui-même.

Les deux autre méthodes se basent sur l'API native du navigateur et ne référencent à aucun moment l'objet `Component`. Ces méthodes n'enregistrent ainsi pas le composant au niveau du navigateur. Dans une telle situation, la fonction `Component.register` peut être utilisée pour forcer une référence vers l'objet `Component`. Un nombre arbitraire de composants peuvent être passés à `register`.

```
Component.register(HelloWorld, AnotherComponent, ...)
```

Dans le cas des composants utilisés dans le template d'un autre composant, ces composants doivent être explicitement spécifiés dans la liste de dépendances du composant de premier niveau. Ainsi, référencer l'objet `Component` de niveau supérieur référencera également tous les composants des niveaux inférieurs et les enregistrement seront correctement effectués.

```
object HelloWorld extends Component[HelloWorld](
   ...,
   dependencies = List(One, Two, Three)
)
```

3.3.2 Element

La définition du comportement d'un composant se fait par la définition d'une sous-classe de `x.c.Element` qui est ensuite passée au constructeur de `Component` (§ 3.3.1). Chaque instance du composant sera alors une instance de cette classe.

Un élément hérite de tous les éléments nécessaires à la définition d'un comportement de composant par le biais de la classe `x.c.Element`. Le seul paramètre restant à spécifier est le composant qui est implémenté par cet élément, ce qui est effectué par la paramètre passé au constructeur de `Element`.

```
1 class HelloWorld extends Element(component = HelloWorld)
```

La déclaration ci-dessus est donc suffisante pour la définition d'un comportement de composant valide. La classe `Element` offre de nombreux outils aux instances de ses sous-classes.

Interface `HTMLElement`

`Element` hérite de l'interface `HTMLElement`, définie par le navigateur et les standards web. Cette interface hérite à son tour d'un ensemble d'autres interfaces telles que `dom.Element` [3], `dom.Node`, `EventTarget`.

Une instance de `Element` possède donc les méthodes et attributs usuels des éléments HTML, tel que par exemple `style`, `parentNode`, `querySelector` ou `addEventListener`. C'est aussi un argument valide pour les méthodes de manipulation du DOM telles que `appendChild` ou `replaceChild`.

La chaîne d'héritage d'un `Element` Xuen est alors compatible avec les standards web, qui impose à la classe utilisée lors de la définition d'étendre `HTMLElement`.

ShadowRoot

Xuen se base sur les mécanismes du *Shadow DOM* pour implémenter templates et feuilles de styles. Chaque instance d'un composant est ainsi associée à un sous-arbre Shadow DOM, accessible depuis l'attribut `shadow` défini par la classe `Element`.

Cet attribut est similaire à l'attribut `shadowRoot` définie par la spécification Shadow DOM à la différence que `shadow` est garanti d'être non-nul. Un sous-arbre Shadow DOM est automatiquement construit lors de l'accès à l'attribut si celui-ci n'existe pas. Un sous-arbre Shadow DOM est également automatiquement construit si le composant a défini un template ou une feuille de style.

Il est généralement déconseillé de manipuler le sous-arbre Shadow DOM manuellement. Ceci est en général effectué de façon déclarative à partir du template. Il peut cependant être nécessaire d'accéder à l'instance d'un élément présent dans le sous-arbre à partir de l'implémentation du composant.

3. La spécification DOM définit également une interface `Element`, à distinguer de la classe abstraite `xuen.component.Element` qui est une implémentation spécifique de cette interface. En pratique, l'interface `x.c.Element` est rarement utilisée par le code utilisateur, et l'interface DOM est généralement utilisée en tant que `dom.Element` en Scala.js

Dans une telle situation `querySelector` peut être utilisé à partir de `shadow` pour accéder aux éléments du sous-arbre.

```
class HelloWorld extends Element(HelloWorld) {
   private val span = shadow.querySelector("span")
   span.addEventListener("click", ...)
}

object HelloWorld extends Component[HelloWorld](
   selector = "hello-world",
   template = html"""Hello, <span>world</span>!"""
)
```

À noter que, conformément à la spécification Shadow DOM, invoquer la méthode `querySelector` directement sur l'instance `Element` ne permet pas d'accéder aux éléments du sous-arbre Shadow DOM, uniquement aux enfants hors du sous-arbre caché, appelé *light DOM*.

```
// Instancié à partir de :
// <hello-world><span>a</span></hello-world>

class HelloWorld extends Element(HelloWorld) {
   private val a = this.querySelector("span")
   private val b = this.shadow.querySelector("span")
   assert(a.textContent == "a")
   assert(b.textContent == "b")
}

object HelloWorld extends Component[HelloWorld](
   selector = "hello-world",
   template = html"""<span>b</span>"""
)
```

Attributs

Les attributs jouent un rôle similaire aux arguments de constructeur en HTML, ils sont un mécanisme permettant de passer des paramètres à un élément afin de configurer son comportement.

Un élément Xuen peut définir un ensemble de d'attributs auxquels il souhaite avoir accès sous forme d'un signal `Source`. La valeur de l'attribut sera reflétée dans le signal correspondant. Cette association est *live*, c'est à dire que l'attribut sera automatiquement observé et la valeur du signal sera mise à jour si un événement externe venait à modifier sa valeur. Inversement, un changement de la valeur de la source entraînera automatiquement une mise

à jour de l'attribut HTML correspondant.

Un *binding* d'attribut est déclaré en utilisant la méthode `attribute[T]`, produisant un `Source` de type `T` pour l'attribut. Le nom de l'attribut est automatiquement déterminé en fonction du nom de la variable à laquelle cette source est associée.

```
class HelloWorld extends Element(HelloWorld) {
  val foo = attribute[String]
  // Association avec l'attribute `foo`
}
```

Si la détection automatique ne parvient pas à identifier automatiquement le nom de l'attribut en question, une erreur sera générée à la compilation. Il est alors possible de spécifier explicitement le nom de l'attribut. Ceci permet également d'associer un nom différent à la source et à l'attribut manipulé.

```
class HelloWorld extends Element(HelloWorld) {
  val bar = attribute[String]("foo")
}
```

La valeur d'un attribut d'un élément HTML est toujours une chaîne de caractères. Un *binding* d'attribut effectue automatiquement une sérialisation ou désérialisation entre la valeur effective de l'attribut de type `String` et le type `T` utilisé par le signal.

Cette opération nécessite qu'une instance du trait `x.c.AttributeFormat[T]` soit implicitement disponible pour le type `T` en question. Par défaut, les types `String`, `Boolean`, `Char`, `Byte`, `Short`, `Int`, `Long`, `Float` et `Double` peuvent être utilisés pour un attribut.

Propriétés

Les propriétés sont une alternative aux attributs qui ne dépendent pas de `AttributFormat` et peuvent donc être utilisés pour passer n'importe quel type de paramètre à un élément Xuen.

```
class HelloWorld extends Element(HelloWorld) {
  val foo = property[Map[Int, String]]
}
```

En pratique, une déclaration `property[T]` correspond à la construction d'une `Source.undefined[T]`. Cependant, l'usage de `property` souligne l'usage attendu de la source en tant que paramètre de l'élément.

Événements `xuen:connected` et `xuen:disconnected`

La spécification *Custom Elements* définis deux méthodes particulières qui sont invoquées lorsque l'élément est connecté ou déconnecté du document [4] : `connectedCallback` et `disconnectedCallback`.

Puisqu'il n'existe pas de *destructeurs* pour un object JavaScript, contrairement à C++ par exemple, le callback de déconnexion est la meilleure opportunité pour un composant d'effectuer des opérations de nettoyage. Cependant, toutes les déconnexions ne sont pas permanentes et l'élément peut tout à fait être réinséré dans l'arbre du document par la suite. Il peut alors être nécessaire de rétablir les ressources libérées lors de la déconnexion.

Lorsque le système se complexifie, il peut arriver qu'un composant alloue une ressource *à l'intention* d'un autre composant. Celui-ci est alors intéressé de savoir quand libérer cette ressource, c'est à dire lorsque cet élément est *connecté* ou *déconnecté* du document.

Dans l'approche utilisée par le standard, l'élément est le seul récepteur possible des événements et décide les actions à entreprendre. Un composant tiers ne peut pas être averti de ces événements si l'élément ne le prévoit pas. Xuen implémente une approche différente en déclarant les méthodes `connectedCallback` et `disconnectedCallback` comme `final` dans la classe `Element`. Il n'est alors plus possible d'y ajouter d'avantages d'opérations que celles définies par la bibliothèque, y compris pour l'élément lui-même.

À la place, Xuen défini deux événements particuliers `xuen:connected` et `xuen:disconnected` qui sont émis par l'élément lorsque les callbacks standards sont invoqués. De cette façon, un nombre illimité de composant peuvent écouter ces événements et être averti du cycle de vie de l'élément. L'élément lui-même peut également s'enregistrer sur ces événements afin d'effectuer d'avantage d'opérations lors de sa propre connexion ou déconnexion. La figure 3.1 présente un exemple d'utilisation de l'événement par l'élément lui-même.

3.3.3 Template

Un template est une structure de noeuds DOM qui sont automatiquement insérés dans le sous-arbre Shadow DOM d'un composant lorsque celui-ci est instancié. Cette structure est généralement définie à partir du code source

4. Un élément est considéré comme *connecté* lorsqu'il est un enfant de l'objet `Document`. *Déconnecté* dans le cas inverse. En d'autres termes, un élément présent dans la structure DOM d'une page est *connecté* tandis qu'un élément ne pouvant être accédé que par une référence JavaScript est *déconnecté*.

```
class HelloWorld extends Element(HelloWorld) {
  this.addEventListener("xuen:disconnected", { _ =>
    // Libération de ressources attribuée par l'élément
  })
}
```

FIGURE 3.1 – Exemple d'utilisation de l'événement `xuen:disconnected`

HTML correspondant et l'interpolateur `html`, en paramètre au constructeur de `Component`.

Cette structure est *compilée* lors de la création de l'objet correspondant `Template`. Le compilateur va ainsi parcourir récursivement la structure originale afin d'identifier des annotations de *data-binding* associant des comportements particuliers à certains noeuds de l'arbre.

Ces annotations sont fortement inspirées de la syntaxe utilisée par le framework *Angular 2* et prennent la forme d'attributs particuliers placés sur les éléments du template. La comportement exact de ces annotations est spécifié par l'*expression* utilisée comme valeur de l'attribut. La section 3.3.4 détail spécifiquement la syntaxe des expressions. Cette section se concentre sur les annotations disponibles.

Interpolation

Les noeuds `Text` et les attributs d'éléments présents dans le template peuvent contenir des marqueurs d'interpolation `{{ }}`. L'expression contenue dans la double-paire d'accolade sera évaluée puis convertie en `String` avant d'être insérée à la place du marqueur dans le texte final.

```
<div>2 + 2 = {{ 2 + 2 }}</div>
--> <div>2 + 2 = 4</div>

<canvas width="720" height="{{9/16 * 720}}"></canvas>
--> <canvas width="720" height="405"></canvas>
```

Une expression d'interpolation ne peut jamais échouer, les cas de valeurs `null` et `undefined` sont explicitement gérés en retournant le texte correspondant. Dans tous les autres cas, la méthode `toString` est utilisée pour produire la valeur finale.

Il n'est pas possible de placer une interpolation dans un attribut dont le contenu est déjà interprété comme une expression. C'est par exemple le cas des attributs qui sont des annotations de *data-binding*. Il n'est pas non plus

possible d'emboîter une interpolation dans une autre.

Il est cependant autorisé de placer une interpolation dans un commentaire HTML comme utilitaire de développement.

Annotation d'identifiant

Un attribut dont le nom débute par `#` est traité comme une annotation d'identifiant, spécifiant la valeur de l'attribut `id` de l'élément. Cette simple annotation est principalement un sucre syntaxique et la valeur de l'attribut, si elle est spécifiée, est ignorée.

```
<div #foo></div>
--> <div id="foo"></div>
```

Annotation de classe

Un attribut dont le nom commence par `.` est traité comme une annotation de classe. Si aucune valeur n'est fournie pour cet attribut, la classe est inconditionnellement ajoutée aux classes de l'élément.

Si une valeur est fournie pour cet attribut, celle-ci évaluée en tant qu'expression booléenne. Si la valeur évaluée est `true`, la classe est ajoutée à la liste de classes de l'élément. Dans le contraire, la classe est retirée de l'élément. Si cette expression utilise des signaux, ce comportement est dynamique et la classe correspondante sera ajoutée ou retirée au fil du temps.

```
<div .foo .bar="true" .baz="false"></div>
--> <div class="foo bar"></div>
```

Annotation de propriété

Un attribut dont le nom commence par `[` et se termine par `]` est traité comme une annotation d'attribut.

La valeur de l'attribut est évaluée en tant qu'expression et la valeur obtenue est utilisée pour mettre à jour la propriété correspondante de l'élément. Si cette propriété est une `Source`, la valeur de la source est modifiée à la place.

```
<input type="text" [value]="2 + 2">
--> <input type="text"> == $0
--> $0.value == "4"
```

Il est important de souligner que ce type d'annotation n'affecte pas les *attributs* de l'élément mais bien les *propriétés* de l'objet JavaScript correspondant. C'est pourquoi dans l'exemple ci-dessus il n'y a pas d'attribut `value` présent sur l'élément `<input>`, mais `$0.value` retourne effectivement la chaîne de caractères `"4"` [5].

Si aucune valeur n'est fournie pour l'annotation de propriété, la nom de la propriété est utilisée comme expression. Ainsi, une annotation `[foo]` est traitée en tant que `[foo]="foo"`, offrant ainsi une syntaxe raccourcie dans le cas où une propriété d'un élément parent est passée telle quelle à une propriété du même nom dans un élément enfant.

Annotation d'événement

Un attribut dont le nom commence par `(` et se termine par `)` est traité comme une annotation d'événement.

La valeur de l'attribut sera évaluée à chaque fois que l'événement correspondant sera *dispatché* à partir de l'élément. À l'intérieur de cette expression, la variable `event` fait référence à l'instance de l'événement émis. Il est interdit de ne pas spécifier de valeur pour une annotation d'événement.

```
1 <input type="text" (input)="doSomethingWith(event.target.value)">
```

Les événements sont écoutés sur l'élément annoté, et du point de vue de l'élément parent. En d'autre termes, le gestionnaire d'événement est attaché à l'élément lui-même, il n'y a donc pas besoin que l'élément se propage dans l'arbre DOM pour être reçu. En revanche, si l'élément provient du sous-arbre Shadow DOM de l'élément, il est possible qu'il ne traverse pas la barrière du Shadow DOM et soit ainsi invisible à partir de l'élément parent.

Un certains nombre d'événements traversent naturellement la barrière du Shadow DOM, c'est la cas par exemple de `click`, `input` ou `mousemove`. D'autres, comme par exemple tous les événements personnalisés, sont par défaut encapsulés dans le sous-arbre et invisibles de l'extérieur de l'élément. Un tel événement ne pourra être capturé par cette annotation. Dans le cas d'un événement personnalisé, il est nécessaire que le flag `composed` soit défini à `true` lors de la création de l'événement.

5. La notation `$0` est inspirée de l'inspecteur web de Google Chrome, dans lequel la variable `$0` fait référence à l'élément actuellement sélectionné dans l'inspecteur DOM.

Transformation `*if`

Une transformation est une annotation qui modifie dynamiquement la structure du DOM à partir d'expressions.

La transformation `*if` évalue sa valeur en tant qu'expression booléenne. Si la valeur obtenue est `true`, l'élément est inséré dans l'arbre DOM. Si la valeur est `false`, l'élément est retiré de l'arbre et un commentaire est inséré à la place en tant que *placeholder*.

```
<div> <div *if="true"></div> </div>
--> <div> <div></div> </div>

<div> <div *if="false"></div> </div>
--> <div> <!-- *if false --> </div>
```

Transformation `*for`

La valeur de la transformation `*for` doit être un `énumérateur`, une expression particulière définissant les différentes propriétés de l'itération. Pour chaque élément de l'énumérateur, l'élément sera dupliqué et inséré dans l'arbre DOM.

À l'intérieur d'un élément annoté avec `*for`, les variables déclarées par l'énumérateur sont accessibles et correspondent à la valeur courante de l'itération. Il est également possible d'utiliser ces variables pour d'autres annotations présentes sur l'élément, les transformations étant appliquées avant les annotations.

```
<ul> <li *for="i of [1, 2, 3]">{{i}}</li> </ul>
--> <ul> <li>1</li> <li>2</li> <li>3</li> </ul>
```

Précédence des transformations

Si les transformation `*if` et `*for` sont simultanément présentes sur un élément, la transformation `*if` est appliquée en premier, suivi de la transformation `*for`.

L'objectif est d'offrir un mécanisme permettant la désactivation conditionnelle de l'ensemble de l'itération, en traitant `*if` avant `*for`, plutôt que l'élision d'un élément particulier de l'itération, ce qui se produirait si `*for` était évalué avant `*if`. En effet la clause de filtrage `if` de l'énumérateur offre déjà ce mécanisme.

Utilisation combinée des transformations avec `<template>`

Du fait de la syntaxe du langage HTML, il n'est possible d'appliquer une annotation que sur un élément, et non un noeud DOM quelconque. Il n'est par exemple pas possible d'annoter un noeud `Text`.

Dans le cas des annotations basiques, cela n'a généralement pas d'importance, un noeud `Text` ne possède de toutes façons pas d'attribut `id`, `class` ou de propriétés particulières. Il n'émet pas non plus d'événements. Cependant, dans le cas des annotations de transformations, l'impossibilité d'annoter un noeud `Text` peut se révéler gênant.

```
<div><span>...</span> ??*if="..."??text <span>...</span></div>
--> Aucun élément sur lequel placer le `*if`
```

Au autre situation problématique est l'annotation simultanée de plusieurs éléments. Comment faire lorsqu'une transformation `*for` doit produire deux éléments DOM pour chaque élément de l'itération ? Dans l'exemple ci-dessous, chaque *checkbox* est associée à un label.

```
<input *for="i of [1, 2]" type="checkbox" [value]="i">
<span *for="i of [1, 2]">{{i}}</span>
```

Cet exemple produit une liste de 3 *checkboxes* puis 3 labels, ce qui n'est certainement pas le résultat escompté.

Une solution serait d'encapsuler les noeuds à annoter dans un élément neutre tel que `<div>` ou `<span>` et d'annoter cet élément. Cette solution est relativement simple et est généralement la méthode préférée dans ce genre de situation.

Cependant, l'ajout d'un élément supplémentaire peut avoir des effets secondaires indésirables, par exemple au niveau des sélecteurs CSS. Dans le cas d'une itération, ceci n'est pas toujours possible. Dans ces situations, il est possible d'utiliser un élément `<template>` afin d'encapsuler un nombre quelconque de sous-noeuds DOM.

```
<template *for="i of [1, 2]">
   <input type="checkbox" [value]="i">
   <span>{{i}}</span>
</template>
```

Lorsqu'une transformation est appliquée sur un élément `<template>`, cet élément est supprimé du sous-arbre DOM produit par la transformation et seul son contenu est inséré. Il se substitue ainsi à l'élément encapsulant

mentionné précédemment et disparait totalement lors de l'application de la transformation.

Composition DOM avec `<slot>`

Lorsqu'un élément possède un sous-arbre *shadow DOM*, les enfants de son *light DOM*, c'est-à-dire ceux qui ne sont pas dans le sous-arbre caché, sont ignorés et seul le sous-arbre caché est utilisé pour déterminer l'apparence et la structure de l'élément.

À l'intérieur du template, résidant dans le sous-arbre caché, le développeur a la possibilité d'inviter explicitement les éléments du *light DOM* avec l'élément spécial `<slot>`, afin de *composer* les deux arbres en un seul.

Il est par exemple possible de construire un composant "boîte", définissant une structure réutilisable à appliquer autour d'un contenu arbitraire. Le contenu à proprement parler est alors placé dans le *light DOM* de la boîte et sera inclut à la place de l'élément `<slot>` présent dans le *shadow DOM* de cette même boîte.

Ce mécanisme est défini par la spécification *Shadow DOM* et réutilisé tel quel par les composants Xuen. Le document mentionné dans la partie d'introduction sur *Shadow DOM* (§ 3.2) présente l'élément `<slot>` plus en détail, notamment ses interactions avec les feuilles de styles et le concept de *slots* nommés permettant la définition de multiples points d'insertion dans un même template.

3.3.4 Expressions

Les expressions sont à la base des comportements dynamiques dans les templates Xuen. Elles sont fortement inspirées de la syntaxe de JavaScript et suivent en grande partie la sémantique de celui-ci.

Les expressions sont conçues pour être utilisées en combinaison avec les signaux Xuen. Le résultat de l'évaluation d'une expression est transformé en `Signal`, permettant ainsi de profiter du mécanisme de propagation des changements implémentés dans les signaux. Lorsqu'un signal référencé dans une expression change, la valeur de l'expression est recalculée et le template mis à jour en conséquence.

À l'intérieur même d'une expression, les signaux accédés sont décomposés par l'interpréteur. Si l'interpréteur rencontre un fragment d'expression produisant une valeur de type `Signal[T]`, cette valeur sera transformée en simple `T` en invoquant la méthode `value` du signal. Ce processus est récursif, une valeur de type `Signal[Signal[T]]` sera également extraite en une

valeur `T`. Si un signal indéfini est rencontré, la valeur `undefined` est utilisée à la place. L'évaluation de l'expression continue alors avec la nouvelle valeur.

Il n'est donc pas nécessaire lors de la construction d'une expression de se soucier de l'encapsulation d'une valeur dans un signal puisque celui-ci sera automatiquement retiré par l'interpréteur.

```
class HelloWorld extends Element(HelloWorld) {
   val name = attribute[String]
   // `name` est une `Source[String]` une sous-classe de
       `Signal[String]`
}

object HelloWorld extends Component[HelloWorld](
   selector = "hello-world",
   template = html"""
      Hello, {{name}}!
      <!-- Dans une expression, `name` est traité comme une simple
          valeur `String` -->
   """
)

// Utilisation:
<hello-world name="Bastien"></hello-world>
```

FIGURE 3.2 – Exemple illustrant l'extraction automatique des valeurs `Signal` lors de l'évaluation d'une expression

Les expressions disposent également de l'opérateur d'affectation `:=` pouvant être utilisé afin de modifier la valeur d'un signal `Source`. L'expression à la gauche de l'opérateur doit s'évaluer en une instance de `Source` pour constituer un usage valide de l'opérateur. Il ne peut pas être utilisé pour modifier une variable qui n'est pas un signal ou déclarer de nouveau signaux.

Contexte d'évaluation

Une expression est toujours évaluée par rapport à un contexte spécifique. Ce contexte définit le monde extérieur autour de l'expression. Il est par exemple utilisé lorsque l'expression fait référence à des variables ou des fonctions. Dans ce cas, le contexte est invoqué pour obtenir la valeur de cette variable.

Dans la plupart des cas, le contexte d'une expression sera l'`Element` auquel le template est associé. Les variables et les fonctions invoquées correspondent respectivement aux propriétés et méthodes de l'objet `Element`.

```
class HelloWorld extends Element(HelloWorld) {
   val foo = 2
   def bar(v: Int): Int = v * 3
}

<div>{{ foo + bar(5) }}</div>
--> <div>17</div>
```

Certaines constructions, par exemple une transformation `*for` ou une annotation d'événement, peuvent introduire des contextes *enfants*. Ces contextes sont comparables au concept de *scope* d'un langage de programmation. Les propriétés présentes sur le contexte enfant sont disponibles en plus des propriétés du contexte parent. Dans le cas où une propriété est présente à la fois dans le contexte parent et le contexte enfant, la propriété du contexte enfant masque celle du contexte parent (*shadowing*).

```
class HelloWorld extends Element(HelloWorld) {
   val foo = List(3, 5)
   val i = 8
   val j = 13
}

<div>
   <span>{{i + j}}</span>
   <span *for="i of foo">{{i + j}}</span>
</div>
--> <span>21</span> <span>16</span> <span>18</span>
```

Opérateur d'accès protégé `?.`

L'opérateur `?.` est similaire à l'usage à l'opérateur simple `.` : il permet d'accéder à une propriété ou une méthode de l'expression à sa gauche et prend un identifiant à sa droite pour déterminer la propriété référencée.

En revanche, lorsque l'expression à gauche de l'opérateur `.` est `undefined` ou `null`, tenter de l'indexer avec l'opérateur `.` provoque une erreur. Ceci est utile en phase de développement pour identifier les situations où une référence qui ne devrait pas être indéfinie l'est tout de même.

Dans le cas où il est possible et normal que le membre de gauche soit indéfini, l'usage de l'opérateur protégé `?.` ne provoque pas d'erreur et retourne simplement `undefined` à son tour, permettant ainsi de le chaîner autant de fois que nécessaire. Si la référence indexée est définie, l'opérateur `?.` se comporte de façon identique à l'opérateur `.` classique.

3.3.5 Feuille de styles

La feuille de styles d'un composant est un simple fragment de code CSS qui sera injecté dans le sous-arbre Shadow DOM d'un composant. Aucun traitement particulier n'est appliqué au code source CSS.

La feuille de styles est injectée sous la forme d'un élément `<style>` contentant les définitions CSS. Puisque cet élément est placé dans un sous-arbre DOM, la portée de ses sélecteurs est limitée à ce sous-arbre. Il n'y a donc pas de conflits au niveau des identifiants des éléments du template (attribut `id`) ou du nom des classes, conduisant à des sélecteur extrêmement simples et concis.

Spécifiquement l'élément `<style>` est inséré en tant que dernier enfant du sous-arbre Shadow DOM. Ce qui a pour implication que le dernier élément du template ne vérifie pas le sélecteur `:last-child`, puisque l'élément `<style>` est en pratique le dernier élément du template. En revanche, le sélecteur `:first-child` identifie correctement le premier élément dans le template. Si l'usage du sélecteur `:last-child` s'impose, il est possible d'envelopper le template dans un élément *wrapper* et ainsi ignorer la présence de l'élément `<style>` en tant que dernier fils du sous-arbre.

Il est possible de faire référence à l'élément hôte lui même, c'est-à-dire l'élément possédant la racine du sous-arbre Shadow DOM, avec le sélecteur `:host`. La version fonctionnelle de ce sélecteur (`:host(selector)`) permet de faire référence à l'élément hôte uniquement si celui-ci vérifie également le sélecteur passé en paramètre.

L'élément sélectionné par le sélecteur `:host` correspond à l'élément personnalisé lui-même, qui est également accessible en dehors du sous-arbre Shadow DOM. Dans le cas où une propriété CSS est définie à la fois à l'intérieur (par la feuille de styles de l'élément lui-même) et à l'extérieur du sous-arbre Shadow DOM (par une feuille de styles dans le contexte parent), la définition extérieur s'applique et la définition intérieur est ignorée. Par exemple, la marge interne d'un élément `<hello-world>` dont la feuille de styles contiendrait `:host { padding: 12px; }` pourrait être modifiée en spécifiant : `hello-world { padding: 0px; }` dans la feuille de styles de l'élément parent (ou du document).

Ces sélecteurs ne sont pas une spécificité de Xuen, ils font partie de la spécification *CSS Scoping* des standards web. Puisque Xuen se base sur l'implémentation standard des *Web Components* pour ses propres composants, il est possible de profiter des standards annexes développés pour *Web Components* lorsque cela est applicable.

Variables CSS

Le mécanisme de variables CSS est également un standard web qui est à présent implémenté dans la quasi-totalité des navigateurs modernes. En pratique le support des variables CSS est plus répandu que le support de *Shadow DOM* et *Custom Elements*, elles peuvent donc être considérée comme utilisables dans ce projet.

Une variable CSS prend la forme d'une propriété personnalisée dont le nom débute par deux tiret `--`. Par exemple, la déclaration `--foo: 2` défini une variable `--foo` dont la valeur est `2`.

Le nom de *variable* est trompeur. En pratique, ces constructions se comportent plus comme une constante, qui est propagée entre les éléments par le mécanisme de cascade classique de CSS. Cette valeur peut alors être redéfinie et la nouvelle valeur sera effective pour le reste de la cascade.

Ceci peut être comparé à la valeur de la propriété `font-size` qui est appliquée par cascade à tous les sous-élément de l'élément auquel elle est appliquée, et qui peut être écrasée par la définition d'une nouvelle propriété `font-size` sur l'un de ces sous-éléments.

La valeur d'une variable CSS est accédée par la fonction `var()` prenant le nom de la variable en paramètre et optionnellement une valeur par défaut. Par exemple,

```
1 body {
2   --bg-color: blue;
3 }
4
5 div {
6   background-color: var(--bg-color, red);
7 }
```

Les variables CSS sont la méthodes préférée pour paramétrer l'apparence d'un élément personnalisé. La liste des variables CSS utilisée par un élément sont ainsi une part intégrale de l'interface publique de l'élément.

3.3.6 Service

Le concept de Service est utilisé lorsqu'un objet doit être partagé entre plusieurs composants de l'application. Celui-ci est alors créé en tant qu'objet singleton étendant la classe `x.s.Service`.

```
1 object MyService extends Service { ... }
```

Un service peut alors être *utilisé* par un élément en appelant la méthode `Service.use` qui retourne le service passé en paramètre.

```
1 class HelloWorld extends Element(HelloWorld) {
2   val service = Service.use(MyService)
3 }
```

En interne, un service possède un compteur du nombre d'éléments actuellement *connectés* qui utilisent le service. Dès lors que ce compteur est non-nul, le service est `activé`. Lorsqu'il est décrémenté à 0, le service est *désactivé*. Ceci se traduit en l'invocation des méthodes `enable` et `disable` qui doivent avoir été implémentée par le service.

En pratique, la désactivation d'un service est différée de 5 secondes par rapport à la déconnexion du dernier élément afin de laisser l'occasion à un nouvel élément utilisant le service d'être connecté dans l'intervalle et éviter ainsi un cycle désactivation-activation, par exemple lors de la navigation entre plusieurs vues de l'application.

Le concept de service a pour objectif principal de permettre l'implémentation de sources de données pour les composants. Ceux-ci utilisent le service fournissant les informations dont ils ont besoin et le service est en mesure de récupérer ces informations depuis le serveur de façon centralisée entre tous les composants de l'application. Plus spécifiquement, le mécanisme d'activation et désactivation du service a pour but d'offrir à son implémentation une chance de signaler au serveur que les événements de mises à jour des données qu'il fournit l'intéressent ou ne l'intéressent plus.

L'utilisation de la fonction `Service.use` est indispensable. Celle-ci utilise en effet une référence implicite à l'élément à l'origine de l'invocation et se charge d'enregistrer des gestionnaires pour les événements `xuen:connected` et `xuen:disconnected`, fondamentaux au fonctionnement du mécanisme d'activation à la demande des services.

3.4 Solutions existantes

3.4.1 Bindings.scala et Monadic-HTML

Bindings.scala [27] et *Monadic-HTML* [28] sont deux exemples d'implémentation d'un système de template basés sur les concepts de la programmation réactive-fonctionnelle.

Ces deux bibliothèques sont très similaires, *Monadic-HTML* étant initialement inspirée de *Bindings.scala*. Parmi les différences notables, on peut noter

l'approche basée sur les macros compilateur utilisée par *Bindings.scala* et la ré-implémentation des classes XML de Scala dans *Monadic-HTML*.

Dans les deux cas, l'objectif est de transformer le code du développeur composé sur la base des littéraux XML intégrés au langage Scala en une structure dynamique qui peut être insérée dans l'arbre DOM du document.

Les littéraux XML, initialement ajoutés au langage Scala alors que l'usage de XML était beaucoup plus prévalent qu'il ne l'est aujourd'hui, sont aujourd'hui considéré comme une complexité excessive du langage et sont sur le point d'être retirés en faveur d'un interpolateur `xml`. Dans le cas de *Bindings.scala*, l'implémentation sous forme de macros devra être ré-implémentée pour supporter la nouvelle syntaxe proposée.

Dans le cas de *Monadic-HTML*, l'approche n'est tout simplement plus valide. La bibliothèque utilise en effet une implémentation alternative des classes XML utilisées par Scala pour matérialiser les littéraux XML à l'exécution, en principe disponibles dans une bibliothèque indépendante. Puisque la syntaxe n'est que sucre syntaxique au niveau du compilateur, les nouvelles classes dont la sémantique est considérablement différentes de l'implémentation originale sont utilisées aveuglément par le compilateur et conduisent au résultat désiré.

Le retrait du support de la syntaxe XML au niveau du compilateur Scala serait synonyme de retrait de l'implémentation des sugres syntaxiques sur lesquels *Monadic-HTML* est construit, sans offrir d'alternatives évidentes puisque le mécanisme était un *hack* dès le début.

Dans le cadre du développement de Xuen, le choix a été fait de mettre de côté les littéraux XML, pour se concentrer sur une implémentation basée sur les interpolateurs. Par conséquent, une partie du travail précédemment effectué par le compilateur Scala au moment de la compilation est à présent effectué au *runtime* dans le navigateur. La disponibilité du DOM et du parser HTML du navigateur est un avantage au niveau de l'implémentation, mais les performances s'en trouvent inévitablement impactées. L'approche n'exclut pas une utilisation future de macros pour effectuer la compilation du template au moment de la compilation du code.

Une autre différence notable entre ces bibliothèques et Xuen est le niveau d'encapsulation utilisé. *Monadic-HTML* et *Bindings.scala* produisent des morceaux d'arbre DOM qui sont ensuite insérés dans le document, sans encapsulation particulière, notamment en ce qui concerne l'application de styles CSS. Xuen propose dès le départ une abstraction sous forme de *composants* indépendants et isolés construit sur la base des derniers standards du web.

L'implémentation des signaux de ces deux bibliothèques diffère légèrement

de l'implémentation qui en est fait dans Xuen. Les signaux sont manipulés explicitement de façon monadique , il n'existe pas de signaux pouvant représenter des expressions arbitraire. La notion de signaux indéfinis est également absente. Il est nécessaire d'encapsuler explicitement les données manipulées dans une instance de la classe `Option` si le signal peut potentiellement être indéfini, conduisant à une plus grand verbosité à l'utilisation.

3.5 Implémentation

3.5.1 Vue d'ensemble

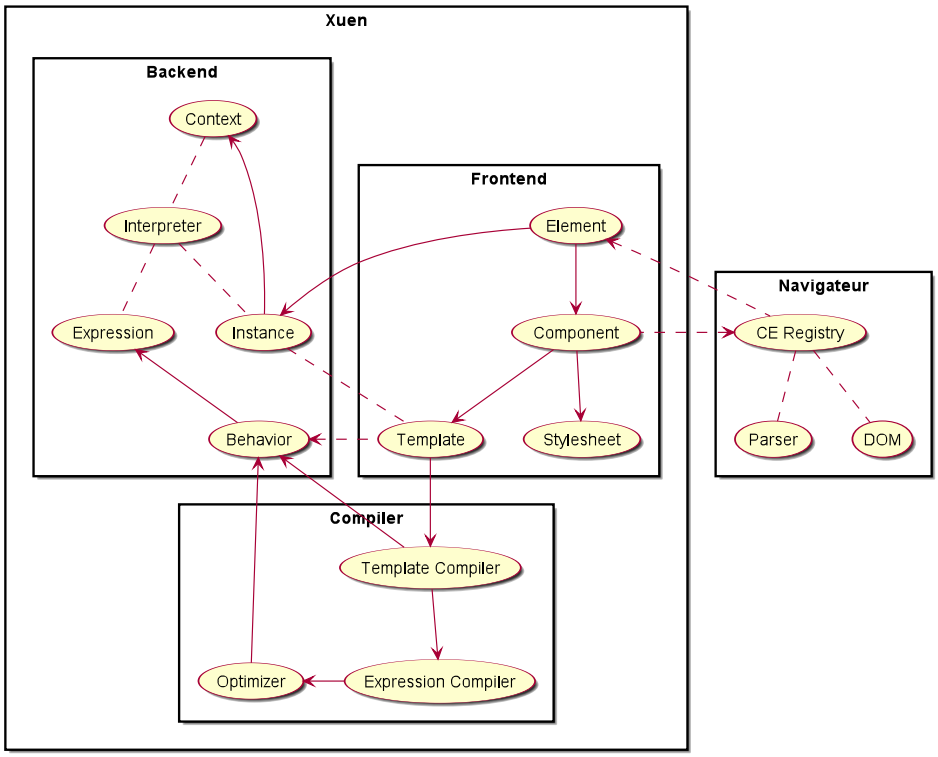

FIGURE 3.3 – Vue d'ensemble de l'implémentation du framework Web

L'implémentation du framework web de Xuen peut être décomposé en 3 grandes parties, tel qu'illustré par la figure 3.3.

1. La partie *front-end* est exposée directement au développeur. Elle est utilisée pour déclarer les composants personnalisés de l'application, leurs comportement et structure.

2. La partie *back-end* est utilisée par le framework lorsqu'un élément est instancié dans le document. Elle fournit les outils nécessaires à l'implémentation des comportements dynamiques du template de cet élément et l'interface avec le système de signaux par le biais des expressions.
3. Finalement le système de compilation est utilisé lors de la définition d'un nouveau composant pour transformer le code source HTML du template en une instance de la classe `Template`, encapsulant à la fois une structure pré-traitée du template, des modèles pour construire les comportements dynamiques de ce template et des expressions sous forme d'arbres syntaxiques prêts à être interprétés.

L'objectif de la partie de compilation est de réduire l'impact du traitement du template au *runtime*. En effet, l'instanciation d'un template se base sur les outils fournis par le navigateur, notamment le parser HTML et son implémentation du DOM afin parcourir et traiter la structure fournie sous forme de code HTML.

En compilant le template et les expressions dès la déclaration, il n'est plus nécessaire de le faire lors de l'instanciation, ce qui permet un investissement fixe au chargement de l'application et des performances accrues lors de son fonctionnement.

Cette structure est en grande partie reprise d'un projet personnel antérieur. La plupart des éléments ont été repensés et réimplémentés, principalement pour les adapter au nouveaux standards *Web Components* version *v1* (par opposition à la version antérieure *v0*). Le parser d'expression a quant à lui été réécrit, passant d'une implémentation fortement inspirée du parser d'expression du framework Angular 2, mais en Scala, à une version développée avec la bibliothèque *scala-parser-combinators* [29], plus propre et idiomatique. La grammaire du langage n'a cependant pas été significativement modifiée. L'optimisateur d'expressions est réutilisé pratiquement tel quel.

3.5.2 Processus de compilation du template

Le template est fourni à la bibliothèque sous forme de code source HTML. La première opération consiste donc à construire dynamiquement un élément `<template>` et à y insérer le code du développeur. Le parser HTML du navigateur sera alors invoqué pour construire une structure de nœuds DOM. Cette structure est ensuite parcourue de façon récursive en commençant à l'élément template créé précédemment.

— Premièrement, la nature du noeud en cours est déterminée. S'il s'agit d'un nœud `Text` ou `Comment`, son contenu est scanné pour y identifier d'éventuelles interpolations.

— Si le nœud est un élément, ses attributs sont observés.

— Si l'élément est annoté d'une transformation `*if` ou `*for`, cette transformation est appliquée.

— Si l'élément possède des annotations de *data-binding*, celles-ci sont traitées.

— Les valeurs des attributs restants qui ne sont ni des transformations ni des annotations de *data-binding* sont scannées pour y identifier des interpolations.

— Une fois l'élément lui-même entièrement traité, l'ensemble de ses enfants est parcouru, réitérant le processus.

Pour chaque nœud DOM devant être associé à un comportement dynamique, un `Behavior` est créé. Celui-ci est identifié par un numéro incrémenté à chaque nouvelle instance. Il sera de modèle pour la construction du comportement dynamique en question. Tous les `Behavior`s créés pour un template son rassemblé dans une `Map` contenue dans l'objet `Template` resultant. L'élément auquel le comportement doit être attaché est quant à lui décoré d'un attribut `xuen:behavior="..."` indiquant l'identifiant du comportement associé à ce nœud.

Si l'élément ne peut pas posséder d'attribut, c'est à dire lorsqu'il s'agit d'un nœud `Text` ou `Comment`, un élément *placeholder* `<xuen:placeholder>` est inséré à sa place dans l'arbre DOM du template et l'attribut est placé sur cet élément. Dans ce cas, lors de la construction du comportement au moment de l'instanciation du template, l'élément *placeholder* sera en premier lieu remplacé par un nœud correspondant à l'original puis le comportement spécifique de ce nœud sera implémenté.

Un `Behavior` peut en réalité implémenter plus d'un comportement pour un même élément. Si deux annotations sont présentes sur un même élément, le `Behavior` construit à l'occasion du traitement de la première annotation est réutilisé pour la deuxième annotation. L'objet `Behavior` sera donc en charge de construire deux comportements dynamiques différents sur le même élément.

À l'inverse de la compilation, le processus d'instanciation est relativement simple. Le template est à présent sous forme normalisée, tous les comportements sont attachés à des éléments annotés par l'attribut `xuen:behavior`. Juste avant d'implémenter les comportements dynamiques, l'objet template est cloné pour construire une nouvelle structure indépendante de l'originale qui est conservée en tant que modèle. La bibliothèque utilise alors la `Map` associant les identifiants des différents comportements enregistrés avec les objets `Behavior` encapsulant la logique de construction pour instancier à proprement parler ces comportements.

Le sélecteur CSS `[xuen:behavior="..."]` est utilisé pour récupérer l'élément sur lequel le comportement doit être appliqué puis la méthode `build` de l'objet `Behavior` est invoquée avec la référence vers l'élément courant. Cette méthode va alors instancier à proprement l'ensemble des comportements liés à cet élément.

De façon générale *instancier un comportement* implique construire une paire signal + observateur qui implémentera le comportement désiré. Cette paire est initialement construite déconnectée. Une fois l'élément hôte connecté, son template est *activé* ce qui entraine la liaison des observateurs avec le signal associé et donc la mise en place du comportement dynamique.

Lorsqu'un élément est déconnecté, son template est *désactivé*. Cette opération déconnecte l'ensemble des signaux et observateurs utilisé dans l'implémentation de ce template, assurant ainsi qu'il n'existe plus de lien entre le reste du système et l'élément déconnecté. Sans cette étape, il existe un risque que les éléments ne puissent pas être désalloués par le *garbage collector* puisqu'une référence subsiste entre le reste du graphe de signaux et eux.

3.5.3 Ordre de construction des *Custom Elements*

Cette section nécessite rédaction

Pour résumer le problème : les éléments customs ne sont "upgradé" (le browser invoque le constructeur de la classe associée) que lorsqu'ils sont "connectés" (leur parent de niveau racine est un Document) SAUF s'il s'agit de l'élément parent étant instancié.

Le problème est que le callback *connectedCallback* est invoqué sur l'élément parent avant que les éléments enfants ne soit considérés comme connectés et upgradés. Ainsi, lorsque le template de l'élément parent est activé avec la connexion au document, les éléments enfants n'ont même pas eu l'occasion de s'initialiser et les mécanismes de data-binding explosent.

La solution adoptée est de "monter" le template de l'élément premièrement dans un sous-enfant bidon du <body>, avant de le remonter correctement dans le sous-arbre DOM de l'élément. Ce passage rapide dans le body du document déclanche l'upgrade immédiat des éléments enfants, mais ajoute son lot de nouvelle complexités. L'élément enfant est en effet considéré comme "connecté" puis "déconnecté" au cours de l'initialisation de son parent, ce qui active le template de l'élément enfant avant que l'élément parent ait eu l'occasion de s'initialiser entièrement !! Afin de prévenir ce comportement, une variable globale `forcedUpgrade` est maintenue pour déterminer si la connexion d'un élément est lié à l'upgrade forcé ou non. Dans le cas où

l'élément est connecté dans le but de l'upgrader, ses callbacks `connected` et `disconnected` sont ignorés.

Jusqu'à présent, cette solution semble fonctionner dans tous les cas. Ci-dessous l'ordre des opérations observées, avec le code de construction correspondant.

```
class Foo extends Element(Foo)
object Foo extends Component[Foo](
   "x-foo", template = html"<x-bar><x-baz></x-baz></x-bar>",
   dependencies = List(Bar, Baz))

class Bar extends Element(Bar)
object Bar extends Component[Bar](
   "x-bar", template = html"<slot></slot>")

class Baz extends Element(Baz)
object Baz extends Component[Baz]("x-baz", template = html"Baz")
```

Constructeur

```
val foo = new Foo
println("--")
dom.document.body.appendChild(foo)
```

```
[x-foo] Upgrade start
[x-foo] Upgrade end
--
[x-foo] Connected
[x-bar] Upgrade start
[x-bar] Upgrade end
[x-bar] Connected
[x-baz] Upgrade start
[x-baz] Upgrade end
[x-baz] Connected
```

Parser HTML (connecté)

```
dom.document.body.innerHTML = "<x-foo></x-foo>"
```

```
[x-foo] Upgrade start
[x-bar] Upgrade start
[x-bar] Upgrade end
```

```
[x-bar] Connected
[x-baz] Upgrade start
[x-baz] Upgrade end
[x-baz] Connected
[x-foo] Upgrade end
[x-foo] Connected
```

Parser HTML (déconnecté)

```
val div = dom.document.createElement("div")
div.innerHTML = "<x-foo></x-foo>"
```

```
[x-foo] Upgrade start
[x-foo] Upgrade end
```

Deuxième partie

Application de démonstration

Chapitre 4

Application de démonstration

Note : Cette section décrit très brièvement l'application GuildTools qui sera développée en utilisant le framework réactif-fonctionnel. La liste de fonctionnalités correspond aux fonctionnalités prévues au moment de ce rapport intermédiaire. Ces fonctionnalités peuvent être amenées à varier d'ici au rendu final.

4.1 Objectif

Application de gestion de guilde sur World of Warcraft.

4.2 Fonctionnalités

4.2.1 Profil de joueur

Ce premier module est destiné à collecter et gérer l'ensemble des données de profil d'un joueur : son nom et prénom, son age, ses informations de contact en jeu ainsi que ses différents personnages.

4.2.2 Calendrier

Un calendrier partagé permettant de définir les soirées de jeu de l'équipe.

Chaque événement offre une liste des joueurs enregistrés comme présents ou absents, ainsi qu'une note optionnelle permettant au joueur d'apporter des précisions supplémentaires.

Un système de gestion des absences de façon globale est aussi disponible, permettant aux joueurs de définir des plages pendant lesquels ils sont indisponibles à tous les événements créés ou futurs.

4.2.3 Roster

Une vue d'ensemble des joueurs du groupe et de leurs personnages avec la possibilité de filtrer selon différents critères.

4.2.4 Whishlists

Un module permettant de saisir les besoins des différents personnages d'un joueur, facilitant ainsi la composition des groupes par les officiers.

4.2.5 Composition de groupes

Un module simplifiant la construction de groupe de jeu en s'assurant qu'un joueur ne se trouve pas simultanément dans plusieurs groupes. Ces groupes sont ensuite exportables vers un événement calendrier.

Chapitre 5

Conclusion

Ce travail a été l'occasion d'explorer le paradigme de programmation réactif-fonctionnel, spécifiquement dans le cadre d'une utilisation combinée avec les technologies les plus récentes de la plate-forme web. La notion de signal est néanmoins un concept très générale, pouvant s'appliquer dans une multitude de situations impliquant la manipulation d'un ensemble de données interdépendantes.

Le projet, initialement orienté sur le développement d'une application complète, a dérivé sur le développement d'une bibliothèque réutilisable et indépendante. Les fonctionnalités de l'application ont donc été réévaluées à la baisse, occupant plus le rôle de projet de démonstration que celui de produit final du travail.

Les résultats sont dans l'ensemble satisfaisants, la complexité et le volume du code produit ne sont pas excessifs par rapport aux fonctionnalités disponibles. L'implémentation est structurée en couches successives construisant sur le concept relativement simple que sont les signaux. La réutilisation des technologies web existantes permet de déléguer une part importante de travail au navigateur.

Le support des technologies liées aux *Web Components* laisse encore beaucoup à désirer au niveau des navigateurs. De même, des constructions introduites par le langage *ES6* depuis plus de deux ans ne sont pas universellement supportées.

Les tendances récentes dans le développement des framworks web tels qu'*Angular*, *Polymer* ou *React* sont une preuve de l'intérêt des développeurs pour des outils permettant la construction de composants isolés, réutilisables et dynamiques. Le support natif des *Custom Elements* et de *Shadow DOM*, sur l'ensemble des navigateurs, serait une étape importante dans l'évolution du développement pour le web.

Concernant plus spécifiquement ce projet, l'exécution de la compilation des templates et des expression au *runtime* laisse beaucoup à désirer. L'utilisation de Scala.js impose, de fait, l'utilisation d'un compilateur pour transformer le code Scala en code JavaScript interprétable par le navigateur. Autant en profiter pour effectuer un maximum de pré-traitements avant de transmettre l'application au navigateur sous forme compilée.

Le système de macro du compilateur Scala est toujours considérée expérimentale depuis son introduction il y a plus de 5 ans avec la version 2.10 du langage. Le développement récent de projets tels que *Scala Meta* [30] et des *new-style macros* proposent des approches plus simples pour le développement de macros et peuvent offrir une piste intéressante à explorer pour le développement d'un compilateur de template sous forme de macros dans le compilateur Scala.

Il est également dommage de disposer d'un langage d'expressions non-typé, dans un framework destiné au langage Scala. L'introduction de l'usage des macros serait également l'occasion de vérifier, lors de la compilation, que le code des expressions soit cohérent avec l'interface déclarée des éléments personnalisés.

Le développement du support du parallélisme dans les graphes de signaux est un autre domaine de développement pouvant étendre l'application du concept de signaux dans d'autres types d'applications, notamment côté serveur et dans un contexte de l'Internet des Objets (IoT).

Finalement, l'écosystème Scala.js est en pleine expansion. Il est important de rester attentif à l'état de l'art en ce qui concerne les frameworks web. L'arrivée prochaine des *Web Component* sera certainement l'occasion de voir se développer d'avantages de bibliothèques exploitant ces outils standards dans leur implémentation.

Troisième partie

Annexes

Grammaire des expressions

Cette section décrit la grammaire des expressions utilisées dans les templates Xuen. La syntaxe utilisée correspond à la syntaxe définie par le W3C pour la description de XML [11]. Les notations `?`, `+` et `*`, indiquent respectivement *0-1*, *1-** ou *0-** répétitions, de façon similaire à leur signification dans les expressions régulières.

La grammaire ci-dessous inclue simultanément les éléments décomposés entre le *lexer* et le *parser* au niveau de l'implémentation. Les productions notées `/.../` correspondent à une expression régulière et sont en principe une règles implémentée au niveau du *lexer*.

Grammaire

```
1  expression ::= <empty> | chain
2
3  enumerator ::=
4     ( identifier "," )? identifier "of" simpleExpr
5     ( "by" simpleExpr )? ( "if" simpleExpr )? ( "with" chain )?
6
7  identifier ::= /[a-zA-Z_$][a-zA-Z0-9_$]*/
8
9  chain ::= ";"* pipe ( ";"+ ( <empty> | pipe ) )*
10
11 pipe ::= simpleExpr ( "|" identifier ( ":" simpleExpr )* )*
12
13 simpleExpr ::= conditional
14
15 conditional ::= logicalOr ( "?" conditional ":" conditional )?
16
17 logicalOr ::= logicalAnd ( "&&" logicalAnd )*
18
19 logicalAnd ::= equality ( "||" equality )*
20
```

```
equality ::=
  relational ( ( "==" | "!=" | "===" | "!==" ) relational )*

relational ::= range ( ( "<" | ">" | "<=" | ">=" ) range )*

range ::= additive ( "to" additive ( "by" additive )? )?

additive ::= multiplicative ( ( "+" | "-" ) multiplicative )*

multiplicative ::= prefix ( ( "*" | "/" | "%" ) prefix )*

prefix ::= ( "+" | "-" | "!" )* secondary

secondary ::= primary secondaryAccess*

secondaryAccess ::= memberAccess | bracketAccess | functionCall
memberAccess ::= unsafeMember | safeMember
unsafeMember ::= "." ( memberWrite | memberRead )
safeMember ::= "?." memberRead
memberWrite ::= identifier "=" simpleExpr
memberRead ::= identifier

bracketAccess ::= "[" simpleExpr "]" ( "=" simpleExpr )?

functionCall ::= "(" callArguments ")"

callArguments ::= ( simpleExpression ( "," simpleExpression )* )?

primary ::= parentheses | selector | literal | reference

parentheses ::= "(" chain ")"

selector ::= simpleSelector | complexSelector
simpleSelector ::= /#[a-zA-Z0-9_\-]+/
complexSelector ::= /@\([^)]+\)/

literal ::= keywordLiteral | numberLiteral | stringLiteral |
    arrayLiteral | objectLiteral

keywordLiteral ::= "undefined" | "null" | "false" | "true"

numberLiteral ::= hexInteger | octInteger | binInteger | decNumber
hexInteger ::= /0[xX][0-9a-fA-F]+/
octInteger ::= /0[oO][0-7]+/
binInteger ::= /0[bB][01]+/
decNumber ::=
    /((0|[1-9][0-9]*)(\.[0-9]+)?|\.[0-9]+)([eE][+\-]?[0-9]+)?/

stringLiteral ::=
```

```
   /:"[^"\\]*(?:\\[\s\S][^"\\]*)*"/ | /'[^'\\]*(?:\\[\s\S][^'\\]*)*'/

arrayLiteral ::=
   "[" (simpleExpression ( "," simpleExpression )* )? "]"

objectLiteral ::= "{" ( objectEntry ( "," objectEntry )* )? "}"
objectEntry ::= objectKey ":" simpleExpr
objectKey ::= identifier | stringLiteral | dynamicKey
dynamicKey ::= "[" simpleExpr "]"
```

Définition théorique des signaux

Cette section, rédigée à l'occasion du premier draft de spécification des signaux, constitue une tentative de définition formelle des signaux. Elle est pour l'instant conservée sous forme d'annexe tandis que la spécification des signaux est maintenant construite à partir de son implémentation concrète.

Les signaux sont des constructions fonctionnelles semi-pures. Puisqu'ils encodent la notion de variabilité, ils sont naturellement dépendant du temps en tant qu'état global. Plus spécifiquement, ils dépendent d'un *indice de génération* spécifique à ce signal désigné par $\gamma_s \in \Gamma_s$ qui est associé à chaque changement potentiel d'état du signal.

Dans le cas d'un signal enfant, l'indice de génération est défini comme un n-uplet constitué des indices de générations de chaque signaux parents sur lesquels il est dépendant. De cette façon, l'indice et donc l'état des signaux parent est encodé dans l'indice du signal enfant et la dépendance vers d'autres signaux ne compromet pas sa pureté :

$$\mathbb{D}^{\gamma}_{child} = \{a, b, \ldots\} \implies \Gamma_{child} = \Gamma_a \times \Gamma_b \times \ldots$$

avec $\mathbb{D}^{\gamma}_{a}$ l'ensemble des signaux envers lesquels a est dépendant pour l'indice de génération γ. Par définition $|\mathbb{D}^{\gamma}_{a}| > 0 \iff a$ est un signal enfant.

Dans le cas d'un signal source, l'indice de génération est une valeur abstraite et distincte pour chaque changement d'état.

$$\mathbb{D}^{\gamma}_{source} = \emptyset \implies \forall \alpha \in \Gamma_{source}, \forall \beta \in Signals, (\alpha \notin \Gamma_\beta) \vee (\beta = source)$$

Un signal peut alors être considéré comme une fonction pure $\Gamma_{sig} \to State_{[T]}$ associant à un indice de génération spécifique un état précis dont la valeur, si elle est définie, est de type T.

$$sig(\gamma_{sig}) \colon \Gamma_{sig} \to State_{[T]} = state \in \{Undefined, Defined_{[T]}(value)\}$$

$$
\begin{aligned}
hold(sig) &: Signal_{[T]} \to Signal_{[T]} \\
&: (\Gamma_{sig} \to State_{[T]}) \to \Gamma_{sig} \to State_{[T]} \\
&= \gamma \mapsto \begin{cases} sig(\gamma) & \text{si } sig(\gamma) \text{ est défini} \\ hold(sig)(\gamma^*) & \text{sinon si } \gamma^* \neq \varnothing \\ Undefined & \text{sinon} \end{cases}
\end{aligned}
$$

FIGURE 1 – Définition de la fonction *hold*

```
def hold[T](parent: Signal[T]): Signal[T] = {
  var previous: Option[T] = None
  Signal {
    val state = parent.option
    if (state.isDefined) previous = state
    state orElse previous
  }
}
```

FIGURE 2 – Implémentation de la fonction *hold*

Les changements d'état d'un signal sont des événements séquentiels. L'ensemble des indices de génération de ce signal forment ainsi un ensemble ordonné sur lequel il est possible de définir la fonction

$$
\begin{aligned}
\gamma^*_{sig} = prev(\gamma_{sig}) &: \Gamma_{sig} \to \Gamma_{sig} \\
&= \begin{cases} \gamma^*_{sig} & \text{si } \exists \gamma^*_{sig}, \forall \gamma^{\alpha}_{sig} < \gamma_{sig}, (\gamma^{\alpha}_{sig} \leq \gamma^*_{sig}) \\ \varnothing & \text{sinon} \end{cases}
\end{aligned}
$$

associant à chaque indice γ_{sig} l'indice γ^*_{sig} associé à l'état qui précédait immédiatement l'état associé à l'indice γ_{sig}. Cette propriété autorise un signal à être défini non seulement en fonction des états actuels d'autres signaux, mais aussi de leurs états antérieurs.

La figure 1 présente la définition d'une fonction *hold* exploitant ce mécanisme pour construire un signal enfant qui maintient sa valeur lorsque son parent devient indéfini.

En pratique, le concept d'indice de génération est implicite. L'accès à un signal se fait toujours à partir de son état le plus récent et la disponibilité des états antérieurs est implémenté en utilisant des variables encapsulées dans le contexte de définition du signal. Les constructions ainsi formées ne sont donc pas strictement pures d'un point de vue fonctionnel mais sont compatible avec la notion théorique d'un signal et ne causent pas de surprise

lors à l'usage. La figure 2 présente une implémentation possible de la fonction *hold* en Scala.

En résumé, les fonctions utilisées pour définir ou transformer un signal doivent être *semi-pures* :

1. elles ne peuvent dépendre d'aucun état implicite, hormis d'autres signaux et leurs états antérieurs, et
2. elles ne doivent pas contenir d'effets de bords observables

De cette façon, l'ordre d'évaluation des signaux ou même leur évaluation différée n'a pas d'importance dans le comportement du système. Ces propriétés ne sont pas vérifiables au niveau du langage, le développeur est ainsi responsable de s'assurer que ses fonctions soient conformes à ces contraintes.

Le concept d'*observateur* (Section 2.2.9) est un mécanisme permettant d'introduire des effets de bords à partir de signaux, de façon sûre et définie.

Opérations élémentaires

Sélection (flatMap)

$$
\begin{aligned}
flatMap(a, f) &: (Signal_{[T]}, T \to Signal_{[U]}) \to Signal_{[U]} \\
&: (\Gamma_a \to State_{[T]}, T \to \Gamma_f \to State_{[U]}) \to \Gamma_{a \times f} \to State_{[U]} \\
&= \gamma \mapsto \begin{cases} f(v)(\gamma_f) & \text{si } a(\gamma_a) = Defined(v) \\ Undefined & \text{sinon} \end{cases}
\end{aligned}
$$

Application (map)

$$
\begin{aligned}
map(a, f) &: (Signal_{[T]}, T \to U) \to Signal_{[U]} \\
&: (\Gamma_a \to State_{[T]}, T \to U) \to \Gamma_a \to State_{[U]} \\
&= flatMap\big(a, v \mapsto \gamma \mapsto Defined(f(v))\big) \\
&= \gamma \mapsto \begin{cases} Defined\big(f(v)\big) & \text{si } a(\gamma) = Defined(v) \\ Undefined & \text{sinon} \end{cases}
\end{aligned}
$$

Filtrage (filter)

$$
\begin{aligned}
filter(a,p)\colon & (Signal_{[T]}, T \to Boolean) \to Signal_{[T]} \\
& : (\Gamma_a \to State_{[T]}, T \to Boolean) \to \Gamma_a \to State_{[T]} \\
& = flatMap\left(a, v \mapsto \gamma \mapsto \begin{cases} a(\gamma) & \text{si } p(v) \\ Undefined & \text{sinon} \end{cases}\right) \\
& = \gamma \mapsto \begin{cases} a(\gamma) & \text{si } a(\gamma) = Defined(v) \text{ et } p(v) \\ Undefined & \text{sinon} \end{cases}
\end{aligned}
$$

Lecture

$$
a.option = \begin{cases} Some(a.value) & \text{si } a \text{ est défini} \\ None & \text{si } a \text{ est indéfini} \end{cases}
$$

$$
a.value = \left(opt \mapsto \begin{cases} value & \text{si } opt \text{ est } Some(value) \\ Nothing & \text{si } opt \text{ est } None \end{cases}\right) \circ a.option
$$

Références

[1] Alex Russell. *Web Components and Model Driven Views by Alex Russell*. Fronteers Conference 2011
`https://fronteers.nl/congres/2011/sessions/web-components-and-model-driven-views-alex-russell`

[2] Alex Russell. *Infrequently Noted*. Blog personnel
`https://infrequently.org/`

[3] Wilson Page. *The state of Web Components*. Mozilla Hacks blog
`https://hacks.mozilla.org/2015/06/the-state-of-web-components/`

[4] Ingo Maier et Martin Odersky, 2012. *Deprecating the Observer Pattern with Scala.React*. EPFL-REPORT-176887
`https://infoscience.epfl.ch/record/176887`

[5] Ingo Maier, 2012. *Scala.react is a reactive programming library for Scala.*
`https://github.com/ingoem/scala-react`

[6] Li Haoyi et al., 2012–2016. *scala.rx : An experimental library for Functional Reactive Programming in Scala.*
`https://github.com/lihaoyi/scala.rx`

[7] HaskellWiki. *Monad laws.*
`https://wiki.haskell.org/Monad_laws`

[8] Evan Czaplicki, 2012. *Elm : Concurrent FRP for Functional GUIs.*
`https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf`

[9] Conal Elliott, 2009. *Push-Pull Functional Reactive Programming*. Haskell Symposium
`http://conal.net/papers/push-pull-frp`

[10] Polymer Project
`https://www.polymer-project.org/`

[11] W3C, 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, Notation
`https://www.w3.org/TR/REC-xml/#sec-notation`

[12] W3C, 2017. *Shadow DOM* (Working Draft)
https://www.w3.org/TR/shadow-dom/

[13] W3C, 2014. *CSS Scoping Module Level 1* (Working Draft)
https://www.w3.org/TR/css-scoping-1/

[14] W3C, 2016. *Custom Elements* (Working Draft)
https://www.w3.org/TR/custom-elements/

[15] Google. *Shadow DOM v1 : Self-Contained Web Components* (Web Fundamentals)
https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom

[16] Scala Native
http://www.scala-native.org

[17] The LLVM Compiler Infrastructure
https://llvm.org/

[18] TypeScript, *JavaScript that scales*
https://www.typescriptlang.org/

[19] ClojureScript
https://clojurescript.org/

[20] Emscripten : *An LLVM-to-JavaScript Compiler*
https://github.com/kripken/emscripten

[21] Google, *What is the Closure Compiler ?*
https://developers.google.com/closure/compiler/

[22] W3C, 2016. *CSS Flexible Box Layout Module Level 1* (Candidate Recommendation)
https://www.w3.org/TR/css-flexbox-1/

[23] W3C, 2017. *CSS Containment Module Level 1* (Working Draft)
https://www.w3.org/TR/css-contain-1/

[24] ScalaTags
https://github.com/lihaoyi/scalatags

[25] ScalaCSS
https://github.com/japgolly/scalacss

[26] Angular
https://angular.io/

[27] Reactive data-binding for Scala
https://github.com/ThoughtWorksInc/Binding.scala

[28] Tiny DOM binding library for Scala.js
https://github.com/OlivierBlanvillain/monadic-html

[29] Simple combinator-based parsing for Scala
https://github.com/scala/scala-parser-combinators

[30] Scalameta
http://scalameta.org/