Développement des outils de dépannage des préprocesseurs OCaml dans l'éditeur *Visual Studio Code*.

Auteur :
Artemiy Rozovyk

Responsable de stage :
Nathan Rebours

Responsable pédagogique :
Christine Tasson

29 août 2021

Table des matières

Chapitre 1

Introduction

Ppx, ou *PreProcessor eXtensions* est une fonctionnalité du langage OCaml, qui permet de transformer l'arbre syntaxique abstrait (AST [1]) vers un autre AST. Après cette transformation, le résultat est proposé au compilateur pour la phase suivante de compilation (analyse sémantique - typage). Il s'agit de ce qu'on appelle la *métaprogrammation*, une technique où un programme traite des données qui sont elles-mêmes des programmes.

Ce genre d'outil se montre particulièrement efficace pour générer du code qui, autrement, serait fastidieux à écrire manuellement. Par exemple, il permet de produire des fonctions de sérialisation d'un certain type de données, ou encore des fonctions d'affichage (*pretty-printers*) [2].

Techniquement, cela est fait en interprétant des parties bien spécifiques de la syntaxe d'OCaml (attributs ou nœuds d'extension) qui définissent la nature des modifications effectuées. En pratique, un ppx est une libraire OCaml, qui prend en entrée un AST correct et qui cherche des éléments qu'elle peut interpréter, puis, génère un nouvel AST (également correct) avec des noeuds insérés ou modifiés.

Par exemple une extension peut prendre la forme suivante :

```
[%addone (1+2)]
```

Dans le code ci-dessus, nous pouvons distinguer deux parties :

- L'identifiant de l'attribut - `addone` qui servira pour identifier le type d'extension et par conséquent la librairie ppx responsable de transformation.
- La charge utile (*payload*), c'est-à-dire l'expression `(1+2)` qui sera réécrite suivant le schéma défini par le ppx.

Logiquement, suite à la transformation (*expansion*) par le ppx approprié, nous aimerions obtenir l'expression : `((1+2)+1)`.

Pour ce faire, un auteur de ppx doit, avant tout, pouvoir établir la correspondance entre le code source et sa représentation avec les types définis dans le module `Parsetree`. Cette capacité lui est nécessaire pour atteindre deux objectifs : Premièrement, il doit pouvoir interpréter le contenu de la *payload*, afin de la traiter correctement (il s'agit le plus souvent de la déconstruction d'une valeur par le filtrage de motif).

En reprenant le même exemple de code, nous pouvons visualiser son AST grâce à ppx_tools\dumpast :

```
ocamlfind ppx_tools/dumpast -e "[%addone (1+2)]"
```

Ce qui donne :

1. *en. Abstract Syntax Tree*
2. **ppx_yojson** et **ppx_deriving.show** sont des exemples respectifs de telles librairies

```
 {pexp_desc =
Pexp_extension
 ({txt = "addone"},
  PStr
   [{pstr_desc =
      Pstr_eval
       ({pexp_desc =
          Pexp_apply
           ({pexp_desc = Pexp_ident {txt = Lident "+"};
             pexp_loc_stack = []},
           [(Nolabel,
             {pexp_desc = Pexp_constant (Pconst_integer ("1", None));
              pexp_loc_stack = []});
            (Nolabel,
             {pexp_desc = Pexp_constant (Pconst_integer ("2", None));
              pexp_loc_stack = []})]);
        pexp_loc_stack = []}, ...)}]);
```

Intuitivement, nous pouvons distinguer un *noeud d'extension* ayant l'identifiant `"addone"`, ainsi qu'un *payload* constitué d'une expression qui est une application de fonction à deux arguments, qui sont des simples constantes.

Deuxièmement, l'auteur d'un ppx doit pouvoir générer des nouveaux noeuds d'AST, notamment en utilisant le module Ast_helper.

Connaissant ces informations, il peut utiliser l'interface d'Ast_mapper afin de définir un ppx (*rewriter*) de manière suivante (exemple simplifié) :

```
1  open Ast_mapper
2  let expr_mapper mapper expr =
3   match expr with
4     | { pexp_desc =
5         Pexp_extension ({ txt = "addone"; loc }, pstr)} ->
6       begin match pstr with
7       | PStr [{ pstr_desc = Pstr_eval (expression, _)}] ->
8               Ast_helper.Exp.apply
9                   (Ast_helper.Exp.ident {txt = Lident "+"; loc=(!default_loc)})
10                          [(Nolabel, expression);
11                           (Nolabel, Ast_helper.Exp.constant (Pconst_integer ("1", None)))]
12      | _ -> raise (Location.Error (Location.error ~loc "Syntax error"))
13      end
14    | x -> default_mapper.expr mapper x;
```

Ce transformateur permet d'interpréter des extensions ayant `"addone"` comme attribut (ligne 5). Ensuite, il récupère l'`expression` de la *payload* (ligne 7) et crée un nouveau noeud - l'application (ligne 8) de `"+"` à la *payload* (ligne 10) et une constante entière de valeur `"1"` (ligne 11). Si la *payload* contient autre chose qu'une expression, une exception indiquant une erreur de syntaxe est levée. La ligne 14 indique que les autres extensions seront traitées par le transformateur par défaut (qui n'est qu'une fonction d'identité en profondeur).

Des outils dédiés à l'établissement de ladite correspondance, bien qu'existants, sont des applications distinctes ou ne sont tout simplement pas mis à jour régulièrement (vis-à-vis de la version d'AST d'OCaml, sujette à de constantes modifications).

De plus, suite à des changements importants, entre autres dans la manière de gérer les nouvelles versions d'AST, les librairies modernes `ppx` se basent sur (ou sont en train de migrer vers) ppxlib, qui propose une interface claire pour l'ensemble des techniques utilisées par les auteurs des ppx. Cette

libraire me servira donc de base afin de réaliser mon travail (sa version d'AST, les fonctions de lecture d'un AST transformé ou encore des fonctions de parcours servant à le sérialiser).

En même temps, un effort important est consenti afin de centraliser les outils standard du développement en OCaml (merlin, dune, opam, ocamlformat...). L'un des exemples d'un tel effort est ocaml-lsp ainsi que l'extension de l'IDE *Visual Studio Code* - vscode-ocaml-platform qui l'utilise.

En continuant cet effort de centralisation, le but de mon travail est donc d'ajouter des fonctionnalités à cette extension, qui doivent permettre d'établir la correspondance entre l'AST (y compris sa version transformée) et le code OCaml (y compris le code résultant de la transformation). Une interface simple d'utilisation et ergonomique doit notamment permettre de visualiser la valeur d'AST d'un morceau de code et inversement, en naviguant l'AST, comprendre quel est le code qui correspond.

Chapitre 2

Contexte

2.1 Tarides

Tarides est une *start-up* fondé en 2018 par les pionniers de langages de programmation et de *Cloud computing* ("informatique en nuage") qui compte une vingtaine d'employées. Les principales axes d'activités de la société sont :

- Le développement d'une infrastructure de création d'applications IoT (*Internet of Things* - l'Internet des objets) décentralisées, sécurisées et peu coûteuses en matière de ressources.
- Des travaux de recherche et de développement sur le projet open-source MirageOS qui a initié et popularisé le développement des *unikernels* - des images pouvant s'exécuter en mode natif sans avoir besoin d'un système d'exploitation.
- Projet open source Irmin, qui est une base de données distribuée ayant des fonctionnalités similaires à `git`.
- Un travail communautaire pour améliorer l'écosystème OCaml qui consiste à créer et maintenir de diverses librairies et outils.

2.2 OCaml Platform

Dans le monde du développement actuel, un langage de programmation ne peut pas être utilisé efficacement sans un ensemble de moyens qui permettent d'effectuer des tâches comme assemblage du projet, génération de la documentation, gestion des dépendances, publication des paquets vers l'extérieur etc. Maintenu par la communauté, *OCaml Platform* répond à ces besoins et représente en soi un ensemble d'outils destinés à augmenter la productivité des utilisateurs du langage OCaml.

2.2.1 Environnements de développement en OCaml

Un développeur passe la plupart du temps devant un IDE[1], qui lui permet de travailler avec le code source. Il peut, entre autres, éditer son code, le formater, l'exécuter ou lancer des testes, chercher la documentation, aller à la définition d'une fonction, et bien d'autres actions.

De plus, étant donné que l'OCaml est un langage de programmation statiquement typé, il profite d'autant plus de l'intégration avec des IDEs. Lorsqu'on développe, voir le type des éléments permet de comprendre plus facilement la signification du programme et d'utiliser cette information pour écrire le reste du code.

Historiquement, OCaml avait un bon support des éditeurs *Emacs* et *Vim*. Cependant, cela impliquait la nécessité de maintenir des *plugins* spécifiques à chaque éditeur (écrits en *elisp* dans le cas *Emacs*, *vimscript* ou *python* pour *Vim*, ou *javascript* pour *Atom*). Chaque plugin devait trouver une manière de mettre à profit tous les outils de l'écosystème OCaml : *merlin* pour l'auto-complétion et l'information relative aux types, *OCamlformat* pour formater le code source d'une manière standardisé, ou encore *Odoc* pour générer la documentation, cf. 2.1.

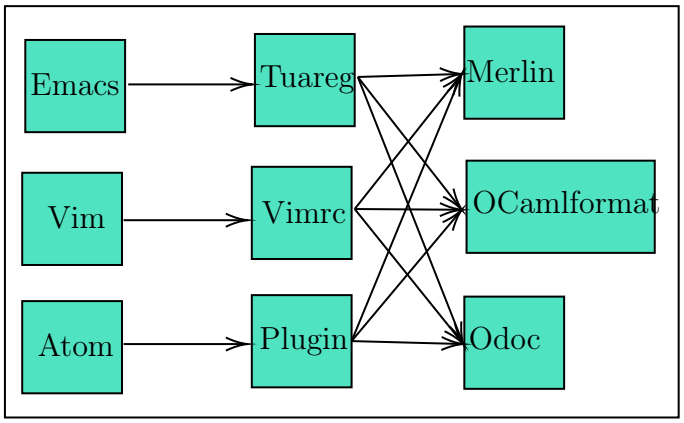


FIGURE 2.1 – Modèle historique du fonctionnement des IDE avec OCaml

Cette approche implique que chaque plugin doit indépendamment refaire un travail similaire d'une complexité importante qui consiste à adapter l'API d'un certain outil du langage à l'interface d'un éditeur particulier. Cela demande beaucoup de ressources en terme de développement et produit un résultat hétérogène (certains éditeurs sont moins bien supportés ou ne sont tout simplement pas à jour vis-à-vis des outils de l'écosystème).

2.2.2 ocaml-lsp

La situation de l'hétérogénéité du support des éditeurs n'est pas propre à l'OCaml, la plupart des langages de programmation faisaient face au même problème. Pour y remédier, un ensemble de développeurs avec Microsoft en tête, ont défini *Language Server Protocol*. Ce protocole permet de s'abstraire des détails spécifiques au langage et proposer une API qui peut être utilisée par plusieurs éditeurs.

En pratique, cela signifie que chaque éditeur possède un client LSP qui communique avec un serveur LSP. Le serveur détecte le langage que le client utilise et c'est donc son rôle de communiquer avec tous les outils spécifiques au langage.

L'idée de cette approche est d'implémenter un seul serveur, qui permettra ensuite aux éditeurs de profiter des fonctionnalités spécifiques du langage (à condition de posséder un client LSP, relativement simple à mettre en place).

1. en. *Integrated development environment* (Environnement de développement)

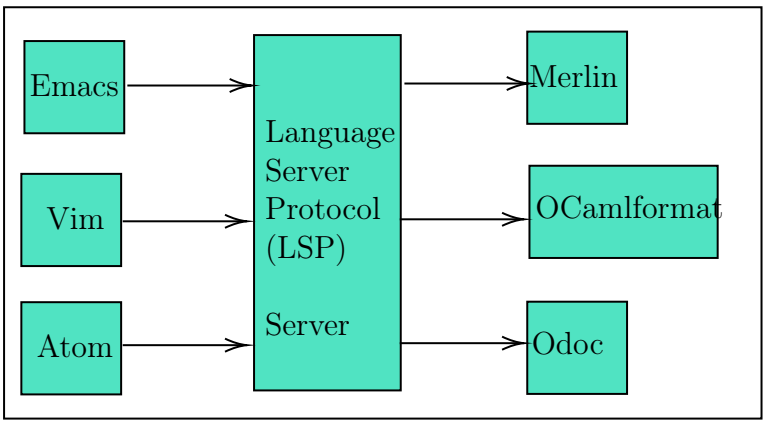


FIGURE 2.2 – Schéma du protocole LSP.

Actuellement, la plupart des langages de programmation implémente un serveur LSP. C'est également le cas d'OCaml avec ocaml-lsp. Comme on peut s'en apercevoir sur 2.2, *ocaml-lsp* se base sur les outils standard du développement OCaml. En cachant la complexité qui se trouve derrière ces outils, il donne une bonne base pour étendre le support d'OCaml dans les IDE modernes. Afin de mieux s'intégrer avec les outils mentionnés ci-dessus, *ocaml-lsp* est entièrement écrit en OCaml.

2.2.3 Visual Studio Code

L'un des premiers éditeurs à supporter LSP était le projet open-source *Visual Studio Code*, développé par Microsoft. Cet éditeur a acquis ces dernières années une popularité considérable. Ce succès est notamment dû à sa fluidité d'utilisation, ainsi qu'à son extensibilité qui rend possibles la définition de toutes sortes de plugins permettant notamment le support des différents langages, intégration avec *git*, débogage, et beaucoup d'autres fonctionnalités très appréciées par la communauté des développeurs.

2.2.4 vscode-ocaml-platform

La communauté OCaml a également développé le plugin vscode-ocaml-platform, qui utilise *ocaml-lsp* et joue donc le rôle de client. De plus, il définit un certain nombre de fonctionnalités comme détection et choix de opam switch (l'environnement contenant les paquets jouant le rôle des dépendances), l'exploration des paquets (et de leurs versions) installés dans l'environnement courant, coloration de syntaxe pour la plupart des types de fichier en rapport avec OCaml, et bien d'autres. De manière générale, le rôle de ce projet est d'utiliser API de VSCode, afin d'intégrer les fonctionnalités (entre autres alimentées via le client LSP) avec l'interface d'utilisateur de l'éditeur.

2.2.5 Utilisation de js of ocaml

Classiquement, une extension *VSCode* est développée en *Typescript*, cette dernière communique ensuite avec l'API VSCode afin de définir les nouvelles fonctionnalités. Cependant, *vscode-ocaml-platform* utilise le compilateur *Js_of_ocaml* qui transforme le *bytecode* OCaml en *Javascript*. Cela est fait dans le but de réduire la non-concordance d'impédance entre ces deux langages. Concrètement, grâce à une bonne intégration de *Js_of_ocaml* avec l'écosystème OCaml, un projet d'une telle complexité peut être entièrement écrit en OCaml et donc profiter de l'excellent *build system* dune. Cela signifie également la possibilité de dépendre sur la plupart des paquets disponibles sur *opam* (à quelques exceptions près), qui seront également compilés vers Javascript. Finalement, le fait que le projet est entièrement écrit en OCaml a également pour le but d'encourager la communauté des développeurs à contribuer.

2.2.6 Ppxlib

Ppx est un système de métaprogrammation OCaml qui permet de générer du code à la compilation. La particularité de ppx réside dans le fait qu'il soit étroitement lié au *parser* d'OCaml : au lieu d'opérer directement au niveau du texte comme certains pré-processeurs dans d'autres langages (expansion de *macros*), il travaille avec la représentation interne structurée - l'arbre abstrait de syntaxe - AST, décrit dans 1. La capacité de travailler directement avec l'AST a l'avantage d'offrir une flexibilité aux développeurs de ppx, qui disposent des informations structurées relatives à la nature du code, ce qui permet d'effectuer facilement toutes les transformations nécessaires.

Cependant, inconvénient du fait de se baser sur l'AST (et *compiler-libs* de manière générale) est le fait que ce dernier est susceptible de changer à chaque fois qu'une nouvelle version du compilateur OCaml voir le jour.

Historiquement, le projet OCaml-migrate-parsetree (OMP) était utilisé pour résoudre ce problème en convertissant les versions d'AST [2]. Récemment, ppxlib a intégré les fonctionnalités d'OMP, en créant OMP2, ce qui avait pour le but de rendre le système plus stable (notamment grâce au fait que ppxlib coordonne son travail avec l'équipe chargée du développement du compilateur). Toutefois, cela à créé 2 mondes séparés : les ppx qui se basaient sur l'ancienne version d'OMP et ceux, utilisant ppxlib. Un effort important est en cours pour porter tous les ppx vers l'utilisation de ppxlib. En ce qui concerne les nouvelles versions de l'AST, qui peuvent toujours impacter le fonctionnement des ppx, l'équipe ppxlib suit les *reverse dependecies* (les ppx qui se basent sur ppxlib) en détectant la présence de dysfonctionnements et propose des patches ciblés afin de les corriger. Cela est fait notamment pour un ensemble de ppx répondant à un certain critère, appelé ppx_universe.

Étant donné l'importance d'AST OCaml pour le monde des ppx, il est primordial d'avoir des outils permettant de l'explorer efficacement.

Chapitre 3

Réalisation

3.1 Exploration d'AST

La première partie de mon stage consiste dans la mise en place d'un ensemble de fonctionnalités au sein du plugin vscode, qui permettraient l'exploration d'AST OCaml.

3.1.1 Choix d'éléments UI VSCode

Dans un premier temps, je me suis familiarisé avec l'API VSCode et les possibilités en terme d'interface graphique qui pourraient servir à afficher l'AST OCaml d'une manière efficace.

— `TreeView`. Il s'agit d'un onglet qui se trouve classiquement à gauche de la fenêtre de l'éditeur texte et qui sert notamment à afficher toutes sortes de données ayant une structure arborescente. L'utilisation principale de ce composant (interne à VSCode) est l'exploration des fichiers du

2. La conversion d'une ancienne version version vers une nouvelle est toujours possible, l'inverse est vrai uniquement pour certaines versions et certaines parties d'AST.

projet. De plus il existe des projets ayant adopté cet élément de UI afin d'afficher l'AST, c'est notamment le cas de vscode-go-ast-explorer qu'on peut voir sur 3.1

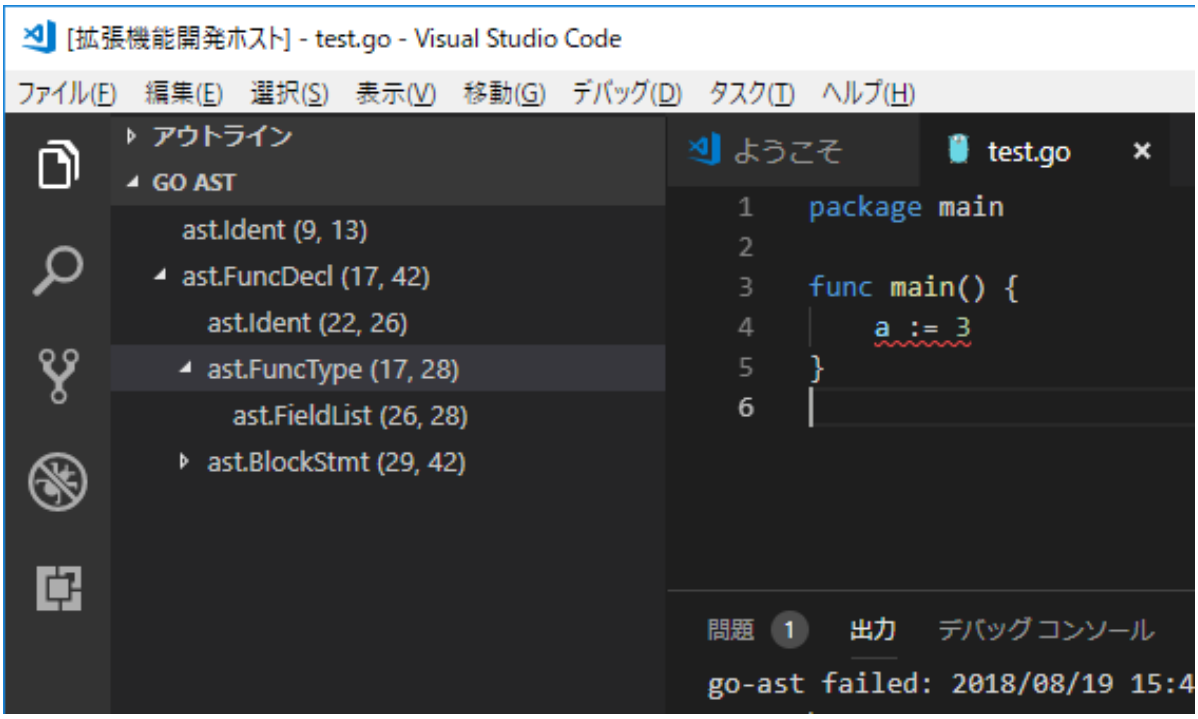


FIGURE 3.1 – Explorateur d'AST du langage Go.

Bien qu'intéressante, la `TreeView` est très limitée en terme de fonctionnalités (difficile de faire la correspondance entre le code et son noeud d'AST). De plus, même un AST d'une (relativement) petite taille dépassera vite les capacités d'affichage de cet élément d'interface, ce qui rendrait la lecture quasi impossible. Cette option a été écartée.

— `Peek` . Cette fonctionnalité correspond à une fenêtre qui s'ouvre à l'intérieur de l'éditeur principal et qui est utilisée notamment pour "jeter un coup d'oeil" sur la définition d'une fonction sans changer de fenêtre cf. 3.2. Il serait intéressant de pouvoir ouvrir cette fenêtre qui contiendrait la représentation d'AST du code correspondant. Malheureusement, malgré de nombreuses demandes (*feature request*) sur github, ce composant UI n'est toujours pas exposé via l'API VSCode.

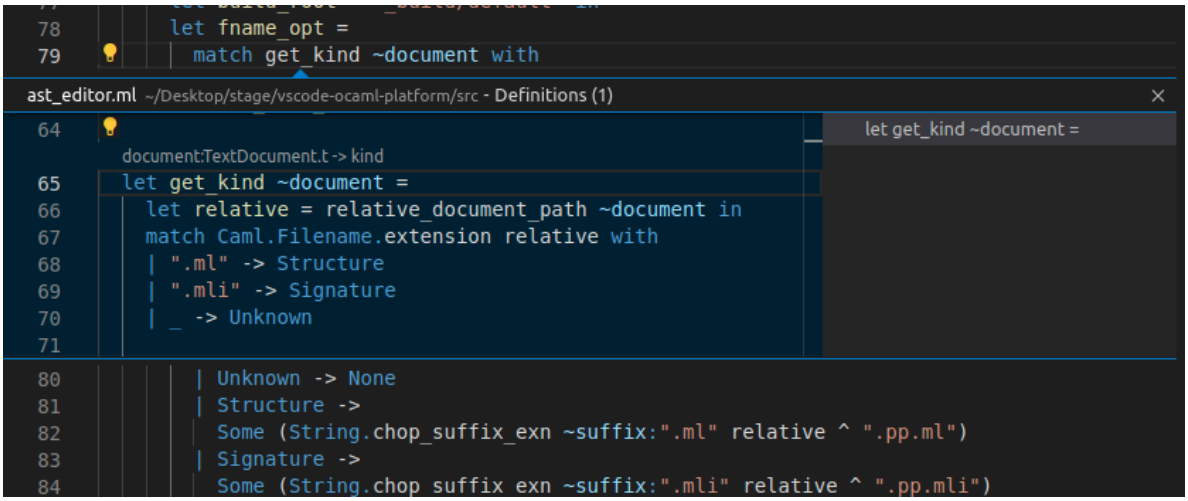


FIGURE 3.2 – Fenêtre interactive (Peek) à l'intérieur de l'éditeur principal.

— `Notebook` À l'instar du Jupyter Notebook, l'API de Notebook offre de multiples possibilités d'affichage du contenu d'un fichier de manière interactive. On pourrait notamment imaginer d'avoir une sorte de fenêtres internes (comme avec les `Peek`) qui seraient disponibles pour chaque `(structure/signature)_item` du fichier. Cependant, au moment de prise de la décision, l'API Notebook était en cours de développement et n'était disponible qu'en version Insiders de l'éditeur (conçue pour tester les nouvelles fonctionnalités). Il n'était donc pas certain que cette fonctionnalité soit mise en production dans un temps raisonnable.

— `CustomEditor` Finalement, la solution qui a été retenue est le `CustomEditor`, qui permet de définir un affichage arbitraire d'un certain type de fichier. L'exemple le plus connu d'utilisation de ce composant (présent dans VSCode par défaut), est le markdown-preview qui permet de visualiser en direct le résultat du rendu d'un fichier d'extension `.md` cf. 3.3. Concrètement, il s'agit d'un mode d'affichage du fichier grâce à un autre composant s'appelant `Webview`. Les `Webviews` fonctionnent de même manière que les navigateurs et ont pour base une simple page HTML. De plus, les `CustomEditors` possèdent un "modèle du document" qui est partagé à travers tous les éditeurs ouverts du même fichier, y compris les éditeurs "classiques". Cela permet de mettre à jour le contenu de l'ensemble des éditeurs lorsque l'un d'entre eux est modifié.

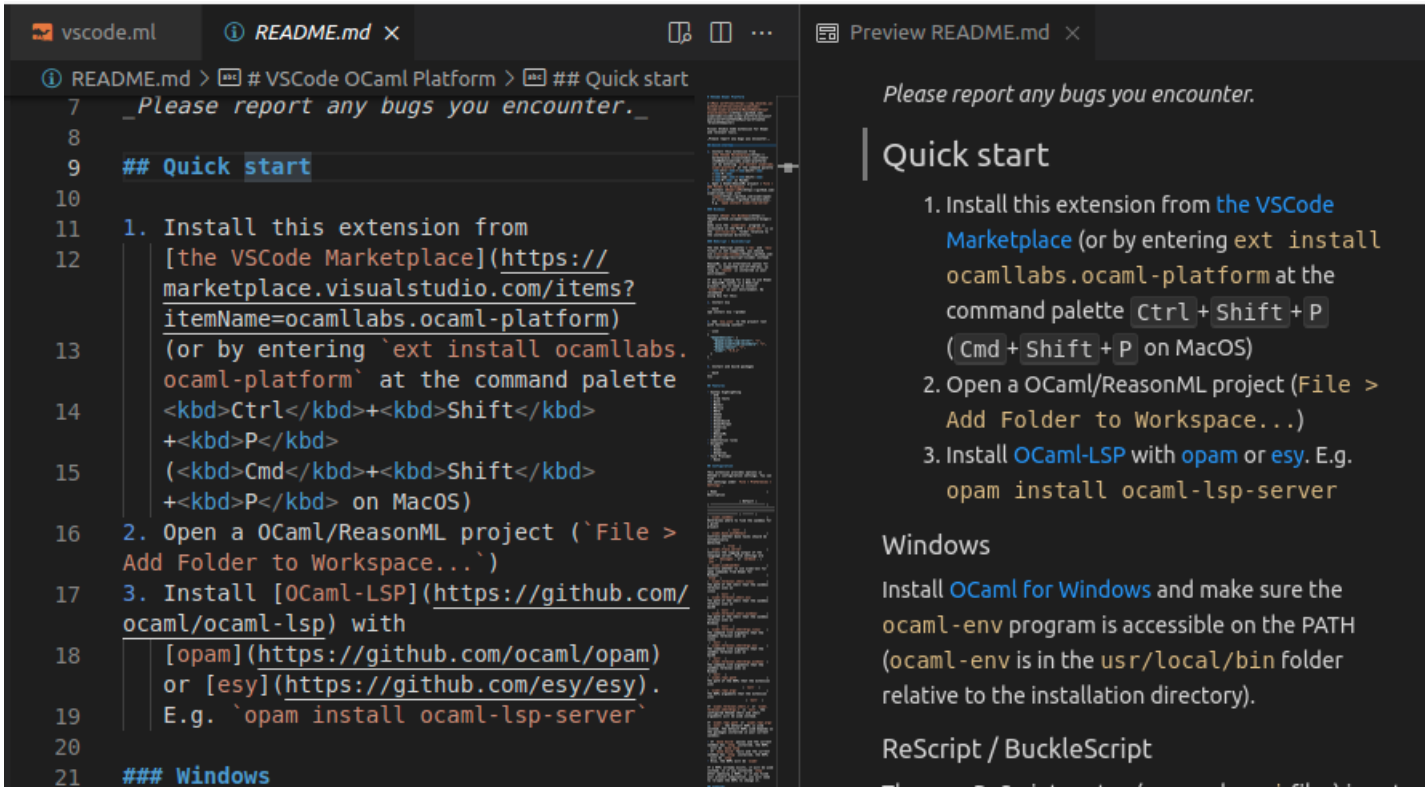


FIGURE 3.3 – Exemple d'un fichier `.md` ouvert à coté d'un `CustomEditor` montrant le rendu final.

3.1.2 Ajout des bindings

Comme j'ai mentionné ci-dessus, l'extension utilise *Js_of_OCaml* afin de pouvoir profiter d'une meilleure intégration avec les outils (écrits en OCaml). Cela est fait grâce à des *bindings* (liaisons) entre le code OCaml et Javascript. Par exemple, étant dans un navigateur quelconque, nous pouvons accéder à la valeur de `window`, de manière suivante :

```
let v = (Js.Unsafe.js_expr "window")##.document
```

En ce qui concerne l'API VS Code, prenons l'exemple de window.createOutputChannel. La définition "manuelle" de *binding* grâce au module `Js_of_ocaml.Js` prendrait la forme suivante :

```
let createOutputChannel ~name =
Js.Unsafe.global##.vscode##.window##createOutputChannel [| Js.string name |]
```

À noter que certaines valeurs OCaml doivent être converties, comme c'est le cas des chaînes de caractères. Cela est dû au fait que les `string` d'OCaml sont des tableaux mutables de caractères à 8 bits, alors que celles de Javascript sont représentés par des chaînes immuables codées sous `UTF-16`.

En théorie on pourrait définir l'ensemble des *bindings* de cette manière. Cependant, il ne serait pas pratique de faire ce genre de conversion pour l'ensemble d'API VS Code. Afin de faciliter la tâche, il est possible d'utiliser une méthode alternative de génération de *bindings* : gen_js_api. Il s'agit d'un outil qui permet de n'écrire que les signatures (en OCaml - `.mli`) pour des librairies Javascript qui seront automatiquement converties en tenant compte des subtilités Javascript (les conventions d'appel, conversion des valeurs, etc.). Ainsi, on exprime `window.createOutputChannel` de manière suivante :

```
val createOutputChannel : name:string -> OutputChannel.t
[@@js.global "vscode.window.createOutputChannel"]
```

Dans cet exemple, *gen_js_ api* va automatiquement convertir le paramètre `name` avec `Ojs.string_to_js` ainsi que `OutputChannel.t_of_js` pour la valeur de retour. [1]

Grâce à un nombre important de *bindings* (environ 30 pourcents de l'API) déjà écrits dans l'extension, j'ai été capable de repérer la manière de faire correspondre la plupart des types de l'API VSCode et donc ajouter le *bindings* nécessaires.

Le point d'entrée des `CustomEditors` est la fonction suivante :

```
vscode.window.registerCustomEditorProvider(viewType: string,
provider: (CustomTextEditorProvider | CustomReadonlyEditorProvider
| CustomEditorProvider), options?: {supportsMultipleEditorsPerDocument: boolean,
webviewOptions: WebviewPanelOptions}): Disposable
```

Dans notre cas d'utilisation, on s'intéresse uniquement au `CustomTextEditorProvider` car nous voulons que notre éditeur reste interactif. La fonction principale de ce module est :

```
val resolveCustomTextEditor :
       t
    -> document:TextDocument.t
    -> webviewPanel:WebviewPanel.t
    -> token:CancellationToken.t
    -> ResolvedEditor.t
    [@@js.call]
```

Elle détermine la nature d'affichage à l'intérieur de la webview, notamment en définissant la valeur de Webview.html.

De nombreux autres *bindings* ont été ajouté au fur et à mesure du développement, notamment des *listeners* comme

- workspace.onDidCloseTextDocument Définissant le comportement à la fermeture d'un document.
- window.onDidChangeActiveTextEditor Changement de fenêtre active.

1. Une valeur OCaml peut être convertie en Javascript s'il possède des fonctionns `*of_js/to_js`.

— window.onDidSaveTextDocument Le comportement à la sauvegarde.

ou encore languages.registerHoverProvider qui définit le comportement quand l'utilisateur survole un morceau de code.

L'ensemble des signatures des *bindings* ajoutés peut être consulté suivant ce lien

3.1.3 Interface utilisateur pour la Webview.

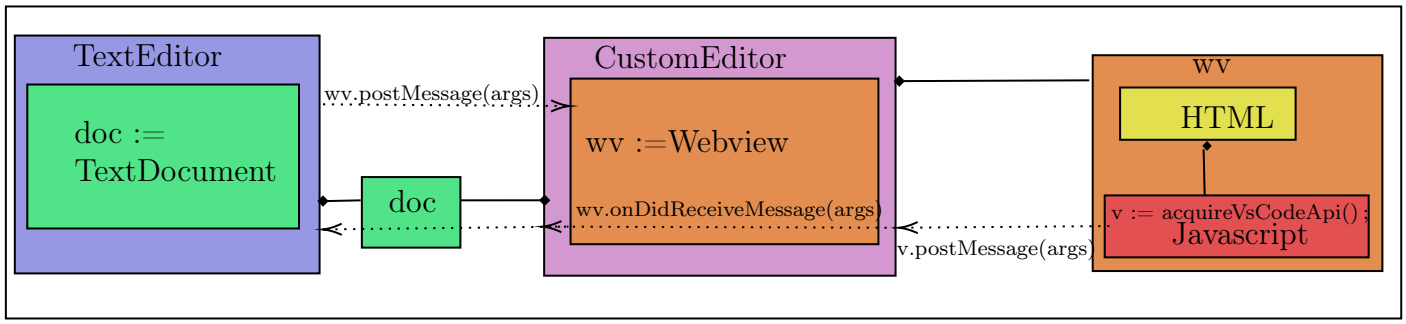


FIGURE 3.4 – Schéma de communication entre l'éditeur VSCode et un `CustomEditor`

Après avoir exploré quelques projets existants qui utilisent l'API de `CustomEditor`, j'ai établi le schéma général du système qu'on peut voir sur 3.4. Lorsqu'un `CustomEditor` est défini, il partage la même instance de `TextDocument` que l'éditeur d'origine. Il peut donc réagir aux changement de ce dernier afin de mettre à jour le contenu de la Webview. Les communications depuis l'éditeur classique se font en envoyant des messages à la `Webview` avec Webview.postMessage. Inversement, lorsque la `Webview` veut communiquer avec l'éditeur principal, elle doit tout d'abord récupérer l'API de VSCode via `acquireVsCodeApi()`, puis envoyer un message à l'instance obtenue. Le traitement de ce message est défini dans onDidReceiveMessage.

Afin de mettre en place toutes les pièces nécessaires, j'ai commencé par écrire une version simplifiée de l'outil. La première étape est l'obtention de la valeur OCaml du type `structure` (alias de `structure_item list`) à partir du code source. Cela est fait en enchaînant les fonctions de la bibliothèque standard suivantes :

— Dans le module `Lexing` :

```
val from_string : ?with_positions:bool -> string -> lexbuf
```

— Dans le module `Parse` de `compiler-libs` :

```
val implementation : Lexing.lexbuf -> Parsetree.structure
```

Ensuite, j'ai écrit un prototype de fonction servant à sérialiser l'AST. Il s'agissait simplement d'une classe qui héritait `Ppxlib.Ast_traverse.fold` qui parcourait l'AST en accumulant les constantes et les opérateurs de fonction.

```
object
  inherit [string list] Ppxlib.Ast_traverse.fold as super

  method! expression e acc =
    let acc = super#expression e acc in
    match e.pexp_desc with
    | Pexp_constant (Pconst_string (s, _)) ->
        ("Pexp_constant ( " ^ "Pconst_string (\"" ^ s ^ "\"))") :: acc
    | Pexp_constant (Pconst_integer (i, _)) ->
```

```
        ("Pexp_constant (" ^ "Pconst_string (\"" ^ i ^ "\"))") :: acc
    | Pexp_ident { txt = Lident op; _ } ->
        ("Pexp_ident " ^ "{txt = Lident \"" ^ op ^ "\"} ") :: acc
    | _ -> acc
```

Le résultat du parcours était simplement une chaîne de caractères qui était envoyée à la `Webview`. Dans ce prototype, l'HTML ne servait qu'à afficher la chaîne de caractères à l'écran. Malgré la simplicité de cette version, une première application fonctionnelle et interactive était en place.

Une fois testé, j'ai procédé à la sérialisation de l'intégralité de l'AST.

Pour être manipulées dans une `Webview`, les informations se trouvant dans `structure` doivent être sérialisées sous format JSON. Cela est possible grâce à la librairie Jsonoo qui permet d'encoder des valeurs de base d'OCaml en une valeur compatible avec `js_of_ocaml`.

Cependant, l'AST d'OCaml est un type très complexe et il ne serait pas pratique d'écrire des fonctions de sérialisation manuellement. Au lieu de faire cela, j'utilise la classe virtuelle paramétrique `Ppxlib.Traverse_ast.lift`[2]. Cette classe définit un certain nombre de méthodes virtuelles (qui doivent obligatoirement être définies) pour les types de base : `char`, `int`, `string`, `list` et ainsi de suite. Pour les autres types, notamment les types algébriques et les enregistrements (*records*) la classe propose des méthodes qui effectuent un parcours structurel en descendant jusqu'aux types les plus élémentaires (gérés par les méthodes virtuelles). Voici un exemple de l'une de telles méthodes :

```
method constant : constant -> 'res=
  fun x ->
    match x with
    | Pconst_integer (a, b) ->
        let a = self#string a in
        let b = self#option self#char b in
        self#constr "Pconst_integer" [a; b]
  ...
```

La valeur de retour de chaque méthode est définie par le paramètre de la classe : dans notre cas il s'agit bien évidemment de `Jsonoo.t`.

L'instanciation de cette classe a la forme suivante :

```
let parse_ast =
object (self)
  inherit [Jsonoo.t] Traverse_ast.lift
  method string value = Jsonoo.Encode.string value

  method float value =  Jsonoo.Encode.float value
  ...
```

L'étape suivante du développement est la définition du contenu `HTML` qui servira à représenter la valeur intégrale de `Parsetree` dans la `Webview`. Tout d'abord, étant donné que l'AST est un type de données très complexe, et que même pour un code relativement simple, la quantité d'information que son AST possède est très importante. Donc, l'une des propriétés les plus importantes de l'interface souhaitée est sa capacité à pouvoir afficher l'AST d'une taille arbitraire.

Tout d'abord j'ai considéré la librairie renderjson qui permet de rendre une valeur JSON pliable : L'utilisation fr `renderjson` convient parfaitement pour réduire la taille d'affichage. Cependant une grande quantité de travail aurait été nécessaire afin de pouvoir naviguer dans l'AST.

Il faudrait notamment définir :

— Un moyen permettant de retrouver rapidement (sans déplier manuellement toute l'arborescence de JSON) le noeud correspondant à un morceau de code.

2. Créée en dérivant la définition du type d'AST via `ppxlib_traverse`

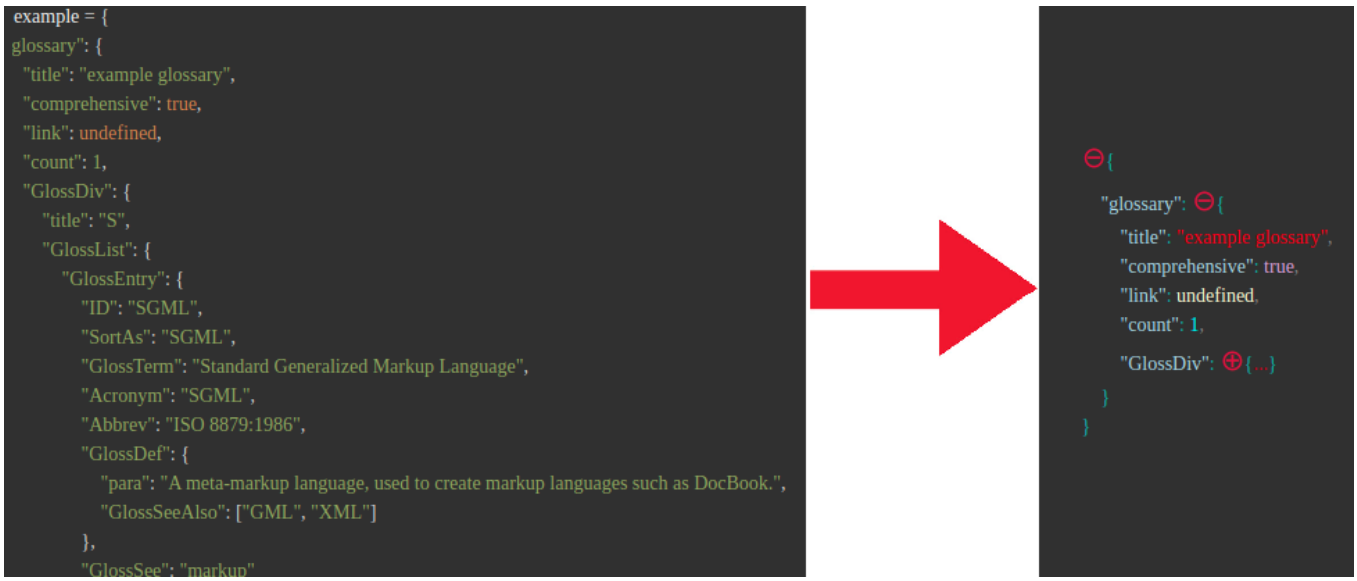
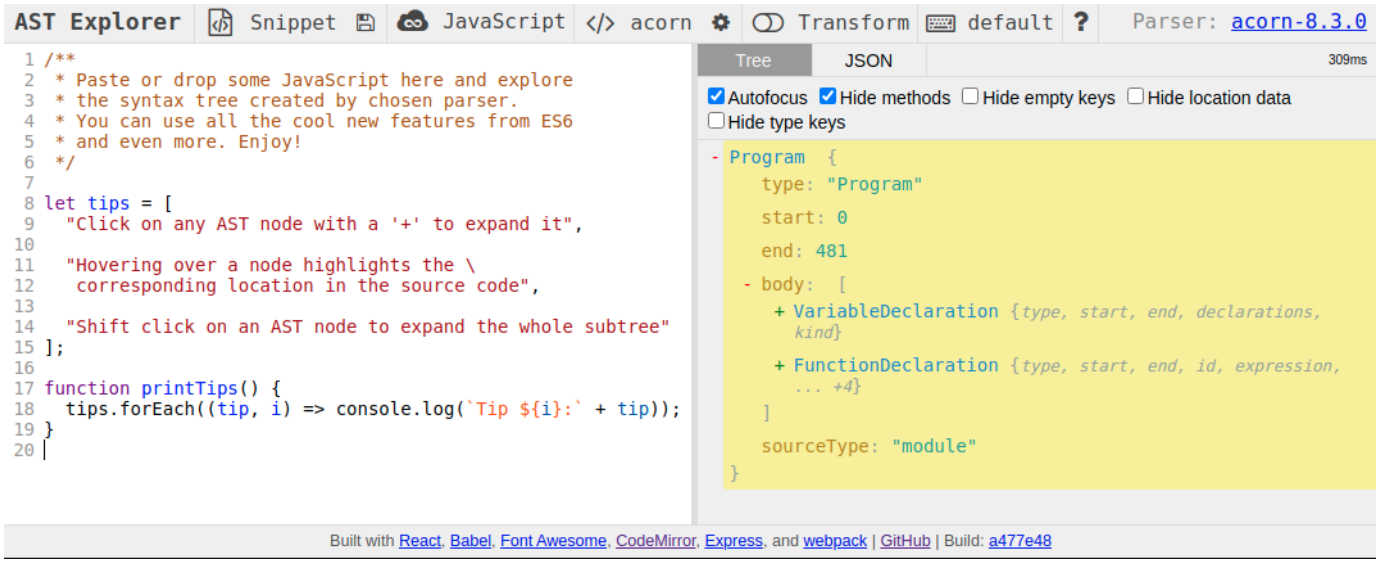


FIGURE 3.5 – Json pliable

— Étant donne un noeud JSON, souligner le code auquel il correspond.
— Mettre en place un algorithme de recherche Location.t dans la valeur de JSON, afin de rendre possibles les deux éléments précédents.

En même temps, un projet qui répond à ces problématiques existe déjà. Il s'agit de astexplorer, qui est un outil générique d'exploration d'AST avec le support de la plupart des langages de programmation. Suite à plus de 7 ans de développement et un effort de 145 contributeurs, cet outil est à la pointe des techniques d'exploration d'AST. De plus, suite à notre discussion avec le propriétaire Felix Kling, le projet a été rendu open-source. Donc, j'ai décidé d'adapter une partie de ce projet afin de l'utiliser comme l'explorateur d'AST OCaml.

Premièrement, l'application d'origine est composé de deux parties :



À gauche on retrouve l'éditeur texte intégré appelé CodeMirror. À droite, la partie d'affichage de la valeur d'AST. Dans notre cas, l'éditeur est remplacé par celui de VSCode. Donc, la première étape consistait à réduire l'application pour qu'elle contienne uniquement la partie d'affichage d'AST, le composant appelé `Tree`. De plus, l'application se base sur le framework React, il est donc nécessaire de compiler le projet séparément, puis de l'intégrer dans la `WebView`.

Étant donné que la `WebView` n'accepte qu'une seule page HTML, l'application doit être compressée

en un seul fichier. Cela est possible grâce à l'outil Webpack, qui construit le graphe de dépendances de l'application et génère un paquet (*bundle*) - une page HTML indépendante contenant le script unifié.

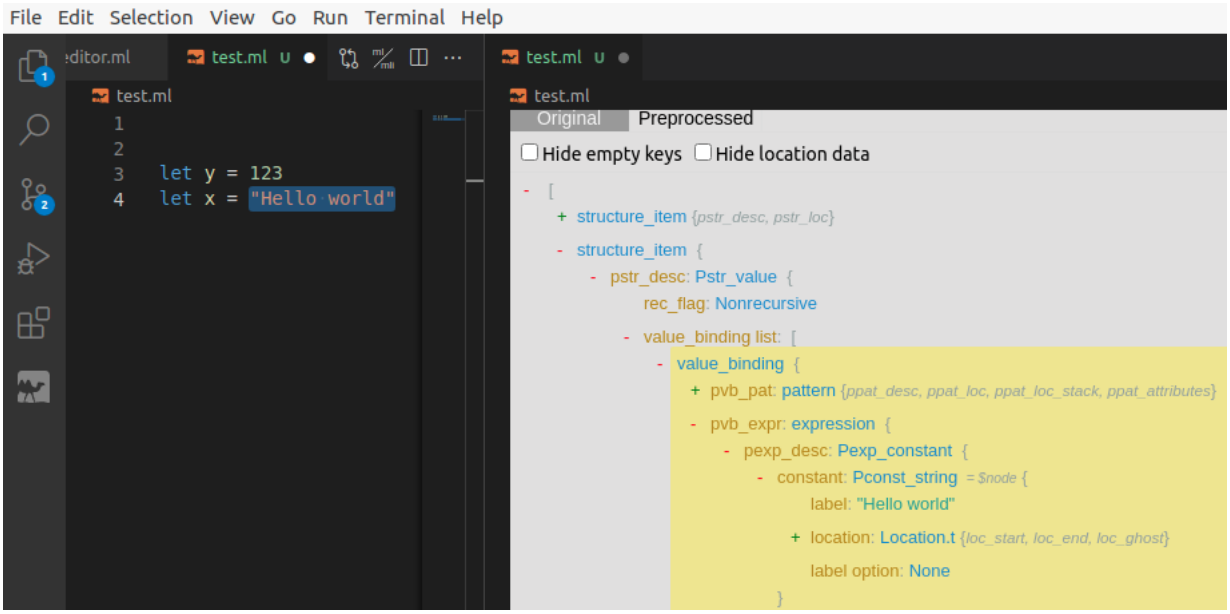


FIGURE 3.6 – Explorateur d'AST intégré à VSCode.

Une fois l'interface intégrée dans la `WebView` 3.6, j'ai procédé à l'analyse des échanges qui sont faits entre `CodeMirror` et l'explorateur d'AST (composant `Tree`) dans l'application d'origine. J'ai ensuite mis en place un système de communication similaire entre l'éditeur VSCode et la `WebView`.

— **Révélation du noeud dans l'explorateur d'AST étant donné un morceau de code.** Étant donné qu'il n'est pas possible de redéfinir le comportement du clic gauche dans l'éditeur standard, j'ai dû définir une commande VSCode qui récupère le décalage (*offset*) de la position du curseur (*caret*) et l'envoie à la `Webview`. Un algorithme est en place qui parcourt la valeur de JSON et cherche le premier *offset* parmi les valeurs du champs `pos_cnum` des `Lexing.position` se trouvant dans `Location.t` :

```
module Location = struct
    type t = Warnings.loc = {
        loc_start : Lexing.position;
        loc_end : Lexing.position;
        loc_ghost : bool;
    }
    ...
end
module Lexing = struct
    type position = {
        pos_fname : string;
        pos_lnum : int;
        pos_bol : int;
        pos_cnum : int; (*offset*)
    }
    ...
end
```

De plus, afin de pouvoir naviguer rapidement dans l'AST, j'ai ajouté un mode où les noeuds AST sont révélés (et centrés) lorsque l'utilisateur survole le code. Il s'agit de la seule manière de récupérer la position du curseur de la souris. Ce mode peut être activé ou désactivé au besoin.

— **Soulignement du code en survolant les éléments dans l'explorateur d'AST.** Lorsque l'utilisateur survole un élément d'explorateur AST, les valeurs `loc_start.pos_cnum` et `loc_end.pos_cn`

sont envoyés à VSCode et interceptés par Webview.onDidReceiveMessage. Ces deux valeurs servent ensuite à appeler la méthode TextEditor.revealRange, ainsi que de mettre à jour le champ `TextEditor.selection`. Ceci permet de souligner et révéler à l'écran (si ce n'est pas encore le cas) le code correspondant.

3.2 Exploration du code transformé et de son AST.

Si l'exploration d'AST est particulièrement utile pour les développeurs de ppx (car elle indique ce que l'outil doit générer), il est également important de pouvoir examiner le résultat des transformations effectuées. Pour cela il faut mettre en place des moyens pour récupérer l'AST transformé par ppx, ainsi que pour pouvoir l'explorer efficacement.

Les transformations ppx sont effectuées par `dune`, qui suit les informations relatives aux transformations (définies dans le fichier `dune` du projet) qui doivent être appliquées. Pendant la compilation, dune appelle le `Driver` de ppx qui effectue la transformation et qui retourne un AST sous un format binaire. Cet AST sérialisé est sauvegardé dans le dossier `_build` avec le même chemin relatif et le nom du fichier (à l'exception de l'extension qui passe de `.ml` à `pp.ml`).

3.2.1 Lecture d'AST transformé.

La première étape est donc la lecture du fichier[3] contenant l'AST transformé. Toutefois, étant donné que nous voulons utiliser la version d'AST de `ppxlib`, nous devons faire cette opération en utilisant la fonction d'API interne `Ppxlib.Utils.Ast_io.read`. Après m'être concerté avec les responsables de la librairie, il a été conclu que cette partie de l'API pouvait éventuellement être exposée. J'ai par la suite ouvert une pull request, où j'ai ajouté un sous-module de `Ast_io` ayant la signature suivante :

```
1  module Read_bin : sig
2     type ast =
3     | Intf of signature
4     | Impl of structure
5
6     type t
7
8     val read_binary : string -> (t, string) result
9
10    val get_ast : t -> ast
11
12    val get_input_name : t -> string
13
14  end
```

La fonction `read_binary` prend un chemin en paramètre et retourne `Read_bin.t`. Nous pouvoir ensuite utiliser l'accesseur `get_ast` afin de récupérer soit une implémentation (s'il s'agit d'un fichier `.ml`), ou une signature (dans le cas de `.mli`). Dans les deux cas, il s'agit d'une valeur du même type `Parsetree` de `compiler-libs`, nous pouvons donc utiliser la même fonction `Ppxlib.Traverse_ast.lift` afin de la sérialiser en JSON. J'ai ajouté par la suite un onglet dédié à l'affichage d'AST transformé. Désormais, l'extension est informée lorsque l'utilisateur change de mode d'affichage ce qui permet d'envoyer la version d'AST approprié.

3. Ce qui est actuellement possible uniquement si la commande `dune build` a été exécuté.

3.2.2 Affichage du code transformé.

Une fois l'AST transformé récupéré, nous pouvons l'afficher comme code source grâce aux *pretty-printers*[4] définies dans `Pprintast`. Pour afficher ce code source, (qui n'existe que pas vraiment en tant que tel) VSCode propose un mécanisme de "document virtuel". Il s'agit d'un éditeur en lecture seule, qui peut être ouvert à partir d'une source arbitraire et qui maintient le support des fonctionnalités standard de l'éditeur, notamment la coloration de syntaxe.

Techniquement, cela est possible grâce à workspace.registerTextDocumentContentProvider qui permet de définir un type de document pour un schéma d'Uri (*Unified Ressource Identifier*) ayant un certain préfixe (dans notre cas le préfixe est `post-ppx`). Ensuite, une commande sert à créer une nouvelle instance du document (en lui associant directement un URI `"post-ppx"`) avec workspace.openTextDocument, replace son contenu avec le code transformé via WorkspaceEdit.replace, puis l'affiche avec window.showTextDocument (variante ayant la signature `showTextDocument(document: TextDocument, options?: TextDocumentShowOptions)` où l'on indique également ViewColumn.Beside, afin que l'éditeur s'ouvre sur le côté de celui d'où la commande est appelée.

3.3 Navigation simultanée

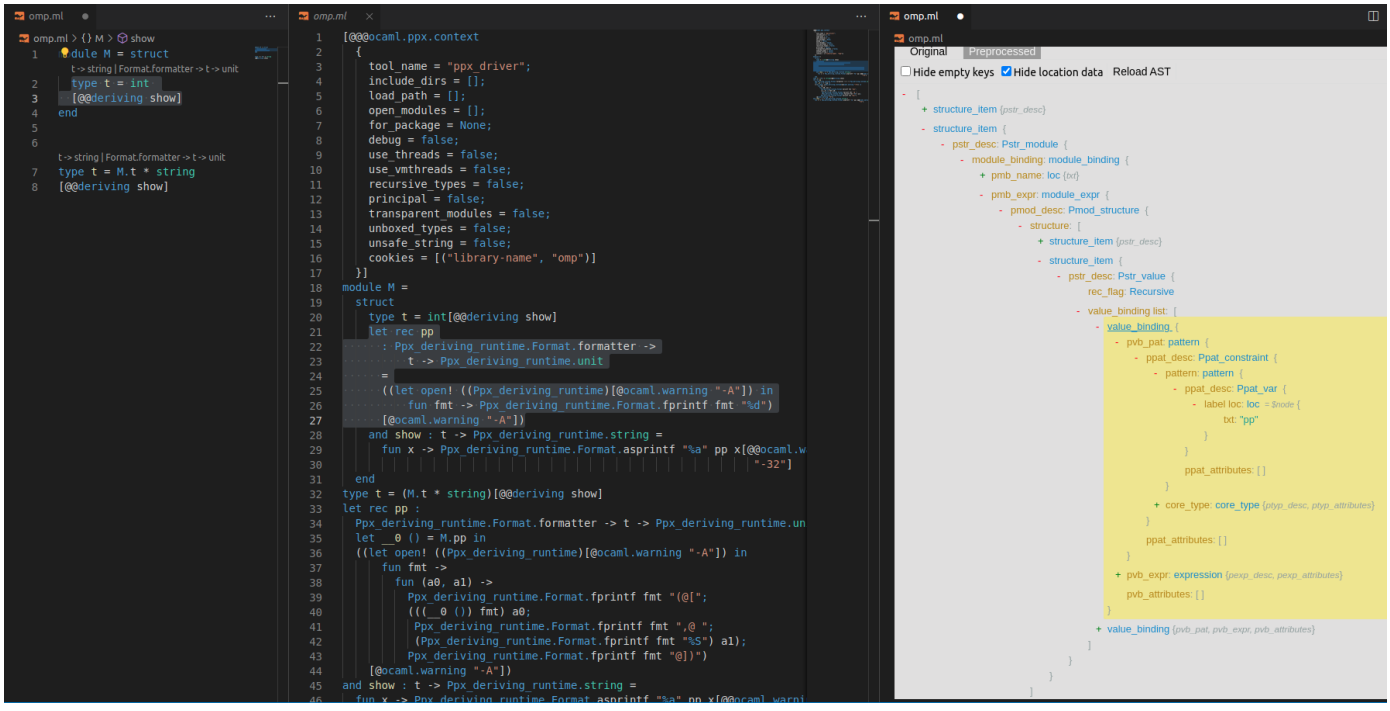


FIGURE 3.7 – De gauche à droite : l'éditeur classique, l'éditeur virtuel contenant le codé transformé, l'explorateur d'AST (en mode AST transformé - à noter le soulignement du code entre lignes 21-27 de l'éditeur du milieu, cela est dû au fait que le curseur survolait le `value_binding` correspondant au moment de la prise de capture d'écran). À noter également le soulignement du code dans l'éditeur tout à gauche qui est à l'origine du code généré (`pp`), cela est possible grâce aux doubles `location` (voir l'explication plus loin).

Après avoir ajouté à l'extension des commandes permettant d'ouvrir (simultanément) les nouveaux éditeurs - explorateur d'AST et du code source transformé, il fallait étendre les moyens de navigation déjà en place, notamment entre l'éditeur virtuel et l'AST transformé.

4. Fonctions permettant d'afficher une valeur de manière structurée.

Afin de pouvoir faire communiquer les 3 éléments présents sur 3.7, j'ai dû établir la correspondance entre les objets qui les représentent : les URI des éditeurs (standards et virtuels) et les `WebView` pour les `CustomEditor`. Pour cela il suffit d'utiliser 2 tableaux associatifs ayant des types : `(String.t, String.t) Map` pour faire correspondre des URI des éditeurs de texte, et `(String.t, WebView.t) Map` pour la correspondance entre l'éditeur standard et un `CustomEditor`.

Finalement, on utilise la fonction de l'API VSCode, qui retourne l'ensemble des éditeurs visibles à l'écran :

```
vscode.window.visibleTextEditors: TextEditor[]
```

Ce qui nous donne tous les éléments nécessaires afin de faire circuler tous les messages.

L'une des spécificités de l'AST transformé, notamment des noeuds nouvellement insérés par ppx, ce sont des informations se trouvant dans les `Location.t` qui font tout simplement référence à l'endroit où se trouve le code d'origine [5]. Cela rend impossible la navigation entre l'AST transformé et l'éditeur virtuel.

La première solution que j'ai adoptée pour résoudre ce problème est de tout simplement faire une passe de *parsing* du code de l'éditeur virtuel (contenant le code transformé) vers un nouvel AST. De cette manière, l'AST qui en résulte contient des "vrais" `locations` qui font référence au code de l'éditeur virtuel.

3.3.1 Non déterminisme dans la représentation du code OCaml

Bien que fonctionnelle, la solution de *reparsing* du code de l'éditeur virtuel pouvait théoriquement être améliorée. L'idée était de garder à la fois les `locations` de l'AST d'origine, ainsi que celles qui résultent du *reparsing*. Cela permettrait, dans un premier temps, de naviguer simultanément dans les deux éditeurs (standard et virtuel) en survolant un noeud d'AST dans l'explorateur. Une autre utilisation potentielle, serait le signalement d'erreurs présentes dans le code généré à l'intérieur de l'éditeur virtuel.

Ainsi, pour obtenir les doubles `locations`, il faudrait parcourir les deux AST en même temps, (de manière analogique à `Ppxlib.Traverse_ast.lift`), et sauvegarder les deux `locations` au moment de les sérialiser en JSON . Cette technique n'est possible que si les deux AST sont structurellement identiques (à la différence près des *locations*), ce qui doit être le cas en théorie. Cependant, après quelques tests, je me suis vite rendu compte que les deux AST ont bien des différences structurelles au-delà des simples *locations*.

Voici un exemple élémentaire d'une telle différence : supposons un `ppx` qui génère un noeud du type `expression_desc` suivant :

```
1  Pexp_tuple [ Pexp_constant (Pconst_char 'a' )  ]
```

L'affichage via module `Pprintast`, nous retournera le code `('a')`, puis, lors du `reparsing` avec `Parse.implementation`, les parenthèses seront ignorées et nous allons obtenir l'expression

```
1  Pexp_constant (Pconst_char 'a' )
```

Cela rend impossible le parcours simultané de deux arbres sans tenir compte de ces différences. Effectivement, le problème semble être connu, mais il n'existe pas de documentation qui indiquerait la nature de telles différences.

5. Cela est intentionnel, car si une erreur est présente dans le code généré, on considère de manière générale que le code d'origine est "responsable" et l'erreur est donc signalée à l'endroit où il se trouve.

3.3.2 ast_diff

Pour identifier les potentielles différences, j'ai créé un outil permettant de détecter les noeuds qui diffèrent et de les visualiser afin de pouvoir analyser leur structure.

L'outil, que j'ai appelé ast_diff, prend en entrée le chemin d'un fichier portant l'extension `pp.ml`. Il essaie de lire le `structure_item list` dans le fichier en question. Si la lecture aboutit, les 2 AST sont obtenus - celui contenu dans le fichier `pp.ml`, ainsi que la version sortie du *reparsing*.

Pour pouvoir afficher les noeuds, j'ai recopié les définitions des types de `Parsetree` pour lesquelles j'ai "dérivé" des fonction d'affichage grâce à `ppx_deriving.show`.

Ensuite, en s'inspirant de la classe `Ppxlib.Traverse_ast.fold`, qui permet de parcourir l'AST en accumulant une valeur (une liste de records par exemple), j'ai créé la version prenant en argument 2 AST au lieu d'un seul. Voici une méthode qui démontre ce fonctionnement :

```
method constant : constant -> constant -> diff list -> diff list =
   fun x x' acc ->
     match (x, x') with
     | Pconst_integer (a, b), Pconst_integer (a', b') ->
         let acc = self#string a a' acc in
         let acc = self#option self#char b b' acc in
         acc
     | Pconst_char a, Pconst_char a' -> self#char a a' acc
 ...
     | _ -> (* Cas de différence *)
         let acc =
           add_diff "constant" (show_constant x) (show_constant x') acc
         in
         acc
```

En utilisant le *patten matching*, cette méthode permet de détecter si les deux types algébriques sont identiques. En cas de différences, (*case* `_`), une valeur du type

```
type diff = {method_name : string; fst_node_pp : string; snd_node_pp : string}
```

est ajoutée à l'accumulateur. Finalement, la liste de `diff` trouvées est parcourue et les différences sont écrites dans les 2 fichiers séparés (qui peuvent ensuite être comparés avec des outils classiques, notamment avec VSCode) : `fst_node_pp` dans le fichier suffixé `*-origine` (car il s'agit du noeud qui se trouve dans l'AST d'origine) et `snd_node_pp` dans le fichier `*-reparsed`. Le préfixe des noms de ces deux fichiers est le chemin d'origine (où les "/" sont replacés par " ?") ce qui permet de garder la trace des fichiers analysés.

3.3.3 Échantillonage

L'étape suivante est la récupération d'un grand nombre de fichiers transformés (d'extension `pp.ml`), afin de les analyser avec `ast_diff`. Afin de couvrir au mieux les potentielles différences qui résultent des transformations faites par des ppx, notamment ceux qui se basent sur *ppxlib*, j'ai procédé de manière suivante :

— En utilisant `opam`, j'ai récupéré la liste des dépendances inverses de `ppxlib` (principalement des libraires ppx) avec

```
opam list --depends-on=ppxlib.0.22.0 --short --columns name>ppxlibrevdeps
```

puis avec la commande similaire, les paquets qui dépendent de l'un des éléments de cette liste :

```
opam list --depends-on='cat ppxlibrevdeps' --short --columns name > res
```

Ce qui nous donne une liste d'environ 500 paquets qui utilisent très probablement un transformateur ppx.

— J'ai ensuite récupéré les sources de chacun de ces paquets avec `opam source`

— Pour construire ces projets, j'ai utilisé opam-monorepo - le plugin d'`opam`, qui permet de récupérer localement (*vendor*) tous les dépendances d'un projet, afin qu'il puisse être construit avec un simple `dune build`.

— Après avoir compilé environ 400 projets, j'ai obtenu 22 mille fichiers d'extension `pp.ml` que j'ai redirigé vers `ast_diff` pour être analysés.

De cette manière, 7881 cas de différences ont été trouvés .

3.3.4 Normalisation

La présence de ce genre de différences est due au fait que le même code OCaml peut avoir plusieurs représentations en terme d'AST. Il existe au moins 11 types de différences comme, (parmi les plus communes) :

```
| Pexp_fun args, Pexp_apply ({ pexp_desc = Pexp_fun args'; _ }, _) ) -> ...
(* 1060 cas détectés *)
```

Donc simplement `Pexp_apply` qui se rajoute lors du *reparsing*. Pour contourner ces différences, j'ai "normalisé" les noeuds qui diffèrent (en enlevant des éléments qui se rajoutent, ou bien en ajoutant ceux qui disparaissent lors du *reparsing*), ce qui a permis de continuer le parcours d'AST. J'ai réussi à éliminer toutes les différences présentes dans mon échantillon. [6] Par exemple, dans le cas de `Pexp_tuple` qui disparaît, je rajoute cette information manquante et je fais un appel récursif :

```
| Pexp_tuple [ exp1 ], pexp_desc' ->
        self#expression_desc
            x (Pexp_tuple [ { exp1 with pexp_desc = pexp_desc' } ])
            acc
```

Tous les exemples des différences trouvées peuvent être consultés en suivant ce lien.

Finalement, ces normalisations m'ont servi pour écrire une version fonctionnelle de `Ppxlib.Traverse_ast.lift2` à deux arguments, mentionné dans 3.3.1. Ce qui permet de faire les doubles *locations* :

```
method! location : location -> location -> (Jsonoo.t, string) result =
  fun { loc_start; loc_end; loc_ghost }
      { loc_start=loc_start'; loc_end=loc_end'; loc_ghost=loc_ghost' } ->
    let loc_start = super#position loc_start loc_start in
    let real_loc_start = super#position loc_start' loc_start' in
    let loc_end = super#position loc_end loc_end in
    let real_loc_end = super#position loc_end' loc_end' in
    let loc_ghost = self#bool loc_ghost loc_ghost' in
    self#record "Location.t"
      [ ("loc_start", loc_start)
      ; ("real_loc_start", real_loc_start)
      ; ("loc_end", loc_end)
      ; ("real_loc_end", real_loc_end)
      ; ("loc_ghost", loc_ghost)
      ]
```

6. Cependant, cela ne garantit pas une couverture intégrale.

Toutefois, il est possible qu'un cas de différence ne soit pas pris en compte. Pour gérer ce cas inconnu (qui passera forcément par `"_"` lors du *pattern matching*), le type de retour a été changé en `('res, string) result`, voici un exemple :

```
method closed_flag : closed_flag -> closed_flag -> ('res, string) result =
   fun x x' ->
     match (x, x') with
     | Closed, Closed -> self#constr "Closed" []
     | Open, Open -> self#constr "Open" []
     | _ -> Error "closed_flag" (* Cas d'erreur *)
```

En héritant cette classe virtuelle, nous surchargeons les méthodes susceptibles de retourner une erreur de manière suivante

```
let with_reparse f on_reparse =
 Ok
   (match f () with
   | Ok res -> res
   | Error m ->
     Vscode.Window.showErrorMessage ~message:
     ("Unable to match the original AST with the reparsed one starting at  " ^ m);
     on_reparse ())

method! closed_flag x x' =
   with_reparse
     (fun () -> super#closed_flag x x')
     (fun () -> parse_ast#closed_flag x)
```

Donc une erreur sera signalé à l'utilisateur et la sérialisation continuera avec la version `lift` à un argument, ce qui résultera en un JSON qui contiendra pas de double `locations` pour le sous-arbre en question.

3.4 Livraison (Pull request)

Étant donné que mon travail se base sur un projet existant, afin de proposer mon travail au projet principal, j'ai été amené à faire une *Pull Request* qui est une demande de *pull* (ajout). Il s'agit d'une fonctionnalité de Github qui permet d'inspecter les changements qui se trouvent sur une branche du projet (généralement par rapport à la branche principale). Cela permet également aux développeurs concernés de discuter de ces changements et d'apporter leur avis.

Donc, j'ai ouvert une *Pull Request* : AST exploration features for ppx developers. #666

3.4.1 Gestion d'érreurs

La plupart des changements qui ont dû être apportés concernaient la gestion d'erreurs. Il faillait distinguer

— Les erreurs qui font partie du flux de travail normal, sont gérés par le type

```
type ('a, 'e) t = ('a, 'e) result =
     |Ok of 'a
     |Error of 'e
```

— Les erreurs qui doivent être signalées dans l'interface d'utilisateur d'explorateur AST - sont simplement un envoi de messages à la `Webview`.

— Les erreurs critiques, où l'on abandonne l'opération sont gérés par une `exception User_error of string`. Cependant, étant donnée que les extensions VSCode sont structurées d'une telle manière, qu'il n'existe pas d'appelant commun à toutes les fonctions où l'exception peut être levée. Nous ne pouvons donc pas mettre en place un `catch` d'exception globale. Pour y remédier, je me suis inspiré de julia-vscode/julia-vscode/pull/1276. J'ai créé un ensemble de fonctions qui permettent d'envelopper les fonctions où l'exception peut être levée.

```
let unpwrap = function
  | `Ok () -> ()
  | `Error err_msg -> show_message `Error "%s" err_msg

let w1 f x =
  try `Ok (f x) with
  | User_error e -> `Error e

let ws f x y =
  match f x with
  | `Ok f' -> (
    try `Ok (f' y) with
    | User_error e -> `Error e)
  | `Error e -> `Error e

let w2 f = ws (w1 f)

let w3 f x = ws (w2 f x)
...
```

Les fonctions qui doivent être enveloppées ont une valeur de retour `unit`, d'où la nature de fonction `unpwrap`. Les fonctions `w{n}` servent à gérer des fonctions d'arité `n`. Par exemple, un `listener` :

```
val onDidReceiveMessage_listener : Extension_instance.t
                                   -> WebView.t
                                   -> TextDocument.t
                                   -> Ojs.t -> unit
```

Est enveloppé de manière suivante :

```
let listener msg =
     Handlers.unpwrap
       (Handlers.w4 onDidReceiveMessage_listener instance webview document
          msg)
   in ...
```

3.4.2 Choix architecturaux

À part d'autres petits changements cosmétiques de la structure du code, j'ai été amené à adapter mon code afin qu'il correspond à l'architecture générale du projet.

— Tous les modules relatifs à ppxlib ont été transférés dans une sous-libraire ppx_tools qui est compilé en mode `byte` (nécessaire pour js_of_ocaml).

— Étant donné que l'extension possédait un état global, toutes les nouvelles références (variables mutables) nécessaires au fonctionnement de l'explorateur d'AST et du code transformé, ont dû être transférés dans un module séparé dont l'instance est attachée à l'état global.

Chapitre 4

Conclusion

Pour conclure, j'ai effectué mon stage de fin d'études de Master 2 Science et Technologie du Logiciel en tant que stagiaire développeur au sein de la société Tarides. Mon rôle était le développement d'un outil permettant l'exploration de la représentation interne du code OCaml - l'arbre syntaxique abstrait, ayant pour le but de faciliter le développement des outils de métaprogrammation *ppx* qui en dépendent fortement.

Lors de mon stage de Master 1, dont le sujet portait également sur la thématique de préprocesseurs OCaml, une grande partie de mon succès était due à l'article de mon maître de stage actuel - Nathan Rebours l'introduction à l'écosystème OCaml ppx (en.) (devenu de facto une référence pour les nouveaux-venus de ppx). Travailler avec lui durant cette mission, ainsi qu'avec la formidable équipe responsable de l'outil principal de métaprogrammation, était un honneur et en même temps une excellente occasion d'apprendre plus sur ppx, OCaml, et le métier d'informaticien de manière générale.

J'ai également pu capitaliser sur mes compétences acquises au cours de ma formation. Les multiples projets que j'ai réalisés en OCaml durant mes études m'ont donné une bonne base de connaissances de ce langage de programmation, qui m'a été précieuse pour comprendre le fonctionnement des projets de taille industrielle. De plus, les technologies *web* qui m'ont été enseignés tout au long du Master STL étaient indispensables à la réalisation des éléments de l'interface graphique au sein de l'éditeur *Visual Studio Code*.

Au final, mon travail contenant l'ensemble des nouvelles fonctionnalités a été inclus dans la branche *git* principale de *vscode-ocaml-platform* et sera publié dans une prochaine version de l'extension. Je suis très fier d'avoir pu contribuer à ce projet qui est utilisé par un grand nombre de développeurs OCaml.

Fort de cette expérience, j'aimerais par la suite continuer sur cette voie et poursuivre une carrière de développeur en OCaml.