Programmer des chatbots en OCaml avec Watson Conversation Service

Guillaume Baudart, Louis Mandel et Jérôme Siméon

IBM Research AI

Abstract

Les chatbots sont des applications avec une interface conversationnelle. Ils sont composés d'une partie application et d'un moteur de conversation. Dans cet article nous introduisons la bibliothèque `wcs-ocaml` qui permet de programmer les deux parties d'un chatbot en OCaml en se reposant sur le moteur de conversation *Watson Conversation Service* (WCS).

1 Watson Conversation Service

De plus en plus d'entreprises délèguent leur services clients et une partie de leur support technique à des agents conversationnels ou chatbots. Ces chatbots communiquent avec l'utilisateur en langage naturel. Grâce à des avancées techniques récentes, les chatbots sont de plus en plus largement adoptés [14]. Ils peuvent être utilisés dans une page web, dans un système de messagerie, voire au téléphone et sont programmés pour répondre à des questions courantes, aider à la navigation d'un site web, ou pour guider le remplissage de formulaires en ligne.

La Figure 1 présente l'architecture générale d'un chatbot programmé avec le service de conversation d'IBM Watson$^{\text{TM}}$: *Watson Conversation Service* (WCS) [8]. La plupart des services récents de développement de chatbots utilisent une architecture similaire [20]. Elle est composée de deux parties : une application (interface utilisateur, logique de contrôle, orchestration) et un moteur de conversation.

Le moteur de conversation implémenté par WCS reçoit des entrées de l'utilisateur sous forme textuelle. Il analyse ces entrées avec un interprète de langage naturel (*Natural Language Processing*) qui extrait les intentions (ce que l'utilisateur souhaite accomplir) et les entités (les sujets dont l'utilisateur parle). Le résultat de cette analyse est envoyé à l'interprète de dialogue (*Dialog Interpreter*) qui pilote la conversation et décide de la réponse à donner à l'utilisateur.

L'application contrôle les interactions avec l'utilisateur (par exemple, lire les entrées et renvoyer les réponses). Elle est également en charge de maintenir l'état du chatbot. Le service de conversation a une interface fonctionnelle et ne conserve pas d'état entre deux tours de conversation. Enfin, l'application peut appeler des services externes (*e.g.*, service météo) pour

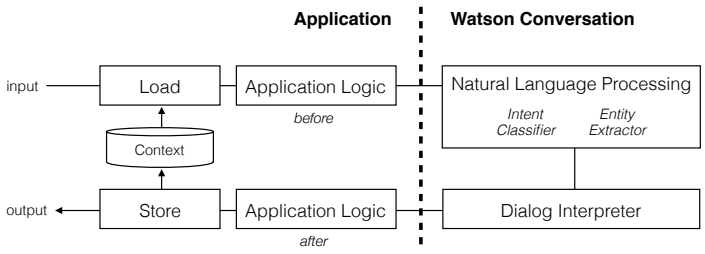

Figure 1: Architecture d'un chatbot programmé avec WCS.

compléter une réponse, réaliser une action (*e.g.*, mise à jour d'une base de données) ou faire des calculs arbitraires.

WCS offre un langage graphique dédié à la spécification des conversations. La partie application appelle le moteur de conversation à l'aide d'une API REST [9] et peut donc être écrite dans n'importe quel langage. Cette approche donne une grande liberté aux programmeurs, mais peut conduire à des difficultés de coordination entre les deux parties d'un chatbot.

Dans cet article, nous présentons `wcs-ocaml` qui embarque WCS dans OCaml et permet de programmer l'intégralité d'un chatbot dans le même environnement. La bibliothèque `wcs-ocaml` est *open source* (https://ibm.github.io/wcs-ocaml) et peut être installée à l'aide du gestionnaire de paquets opam.

Le reste de l'article est organisé de la manière suivante. La section 2 décrit l'intégration en OCaml du langage dédié à la programmation de WCS. La section 3 montre comment appeler le service WCS pour programmer une application de chatbot complète. Les travaux connexes sont discutés dans la section 4 avant de conclure.

2 Spécifier le moteur de conversation WCS

En WCS, les programmes sont appelés *workspaces*. Un workspace est un objet JSON qui inclut les définitions des *intentions*, des *entités*, et du *dialogue*. Pour illustrer ces différentes notions, considérons KnockKnockBot, un chatbot qui raconte des blagues.

> *Bot: Knock Knock*
> *User: Who's there?*
> *Bot: Broken Pencil*
> *User: Broken Pencil who?*
> *Bot: Never mind it's pointless...*

Intentions Les intentions correspondent à ce que l'utilisateur cherche à accomplir. Par exemple, on peut chercher à savoir qui frappe à la porte. Une intention est définie par un ensemble d'exemples qui servent à entraîner l'interprète de langage naturel[1].

```
let who_intent =
  Wcs.intent "Who" ~examples: [ "Who's there?"; "Who is there?"; "Who are you?"; ] ()
```

```
val who_intent : Wcs_t.intent_def
```

Ce type de définition donne une certaine souplesse à l'interprète. Ainsi la phrase *Who's that stumbling around in the dark?* sera associée avec succès à l'intention `who_intent` bien que ne figurant pas dans la liste d'exemples.

Entités Les entités sont les sujets évoqués par l'utilisateur. Dans notre exemple, on peut définir une entité pour les noms des personnages évoqués dans la blague. Une entité peut prendre plusieurs valeurs et chaque valeur possible est associée à une liste de synonymes.

```
let char_entity =
  Wcs.entity "Characters" ~values: [ "Broken Pencil", ["Damaged Pen"; "Fractured Pencil"] ] ()
```

```
val char_entity : Wcs_t.entity_def
```

Pour être détecté par l'interprète, un des synonymes de l'une des valeurs doit apparaître dans le texte soumis par l'utilisateur.

[1] Le code présenté utilise la version 2017-05-26.03 de `wcs-ocaml` ™

Dialogue Le dialogue permet de spécifier le comportement du moteur de conversation par un automate qui réagit aux intentions et entités détectées par l'interprète de langage naturel, et aux informations directement envoyées par l'application. Par exemple la structure de notre blague peut être retranscrite par un automate à trois états : 1) `knock` amorce la blague, 2) `whoisthere` répond à la question avec le nom du personnage, 3) `answer` conclut avec la chute.

```
let knockknock who_intent char_entity answer =
  let knock =
    Wcs.dialog_node ("Knock")
      ~conditions_spel: (Spel.bool true)
      ~text: "Knock knock" ()
  in
  let whoisthere =
    Wcs.dialog_node ("WhoIsThere")
      ~conditions_spel: (Spel.intent who_intent)
      ~text: (entity_value char_entity)
      ~parent: knock ()
  in
  let answer =
    Wcs.dialog_node ("Answer")
      ~conditions_spel: (Spel.entity char_entity ())
      ~text: answer
      ~parent: whoisthere
      ~context: (‘Assoc ["return", ‘Bool true]) ()
  in
  [ knock; whoisthere; answer ]
```

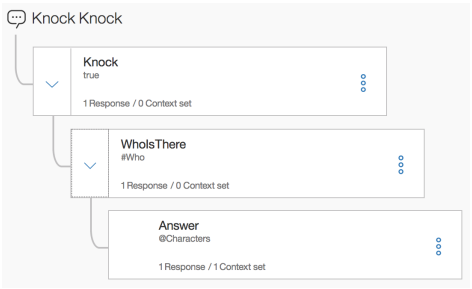

Figure 2: Interface graphique.

```
val knockknock :
Wcs_t.intent_def -> Wcs_t.entity_def -> string -> Wcs_lib.Wcs_t.dialog_node list
```

Les états de l'automate, appelés *nœuds de dialogue* sont construits à l'aide de la fonction `Wcs.dialog_node`. Ce constructeur prend en argument le nom du nœud et un ensemble d'options : la condition d'entrée exprimée dans le langage dédié SpEL [21] (`~condition_spel`), le texte de la réponse du chatbot (`~text`), le père du nœud dans l'automate (`~parent`). Un nœud peut également définir des champs dans l'objet de contexte (`~context`). Cet objet représente l'état du système, il permet de passer des informations entre deux tours de conversation. Ici, le nœud `answer` définit dans le contexte la variable booléenne `return` qui permet à l'application de décider si la conversation est terminée.

La bibliothèque `wcs-ocaml` inclut un module pour construire les expressions SpEL soit comme un arbre de syntaxe, soit directement dans la syntaxe SpEL comprise par WCS [10]. Ainsi, la condition du nœud `answer` peut également être écrite : `~condition:"@Characters"` dénotant une condition qui est vraie lorsque l'entité '`Characters`' est reconnue. L'avantage de la première méthode est de produire une erreur à la compilation si l'entité `char_entity` n'existe pas.

Workspace Nous avons enfin tous les éléments pour définir un workspace complet qui peut être exécuté par Watson Conversation : une liste d'intentions, une liste d'entités et un ensemble de nœuds de dialogue.

```
let ws_knockknock =
  Wcs.workspace "Knock Knock"
    ~intents: [ who_intent ] ~entities: [ char_entity ]
    ~dialog_nodes: (knockknock who_intent char_entity "Never mind it’s pointless") ()

val ws_knockknock : Wcs_t.workspace
```

À partir de cette définition de workspace avec `wcs-ocaml`, il est possible de générer l'objet JSON correspondant et de le charger dans WCS en utilisant l'API ou l'interface graphique. Le workspace peut ensuite être utilisé par une application ou simplement visualisé avec l'éditeur de WCS (figure 2).

Dans `wcs-ocaml`, les workspaces sont des structures de données fortement typées qui sont générées automatiquement avec atdgen [12] en suivant la description des objets JSON manipulés par WCS [9]. La bibliothèque `wcs-ocaml` permet ainsi de manipuler programmatiquement les workspaces.

Le module `Wcs` de construction de workspaces utilise abondamment les labels, ce qui offre une syntaxe plus proche de JSON. Les labels correspondent aux champs des objets JSON dans le workspace et peuvent être donnés dans un ordre arbitraire. Le système de type offre des garanties statiques sur la validité des noms des champs et permet de distinguer les champs nécessaires et optionnels.

De plus, ce module utilise la liaison de nom d'OCaml pour faire la liaison de noms d'objets dans le workspace. Par exemple le lien entre un nœud de dialogue et son prédécesseur (qui est soit le nom du nœud père soit celui du fils précédent) est une simple chaîne de caractères dans le JSON du workspace. En `wcs-ocaml`, le constructeur pour un nœud de dialogue prend en argument un autre un nœud de dialogue correspondant à son prédécesseur, ce qui permet de garantir au moment de la compilation que tous les liens pointent vers des nœuds existants.

3 Programmer une application de chatbot

Le workspace n'est qu'une partie d'un chatbot. Comme illustré sur la figure 1, WCS a aussi besoin d'une partie application. La bibliothèque `wcs-ocaml` permet également d'invoquer un workspace déployé sur WCS avec l'appel REST `message` [9].

Invoquer des workspaces Cette fonction `message` prend en argument le texte de l'utilisateur et le *contexte*, un objet JSON qui contient les informations nécessaires à WCS comme le nom du nœud de dialogue courant. Le contexte peut également contenir les données qui permettent de communiquer entre l'application et le workspace. Par exemple, le nœud `answer` (section 2) définit une variable de contexte `return` pour signaler à l'application que le dialogue est terminé.

À partir de la fonction `message`, on peut définir une fonction `interpret` qui va interpréter un tour de conversation pour un workspace `id` et extraire la valeur du champ `return` du contexte. Pour utiliser WCS, la fonction a besoin des certificats d'authentification du service. Ils sont donnés avec l'argument `creds`.

```
let rec interpret creds id req =
  let resp = Wcs_call.message creds id req in
  let ctx, return = Json.take resp.msg_rsp_context "return" in
  { resp with msg_rsp_context = ctx }, return
```

Le contexte est un objet JSON arbitraire représenté avec la bibliothèque Yojson. Il n'y a donc que peu d'information de typage pour les variables utilisées pour communiquer entre l'application et le workspace. Mais en utilisant `wcs-ocaml` pour construire à la fois le workspace et l'application, il devient possible de partager des structures de données. Dans l'exemple, on aurait pu définir un type `return = { return: bool; }` et utiliser des fonctions de sérialisation en JSON dans le nœud `answer` et de désérialisation dans la fonction `interpret` pour échanger les données de façon plus sure.

Une version étendue de cette fonction `interpret` est fournie par le module `Wcs_bot` de `wcs-ocaml`. En plus d'extraire la valeur du champ `return`, elle permet d'exécuter des sous-dialogues. Les sous-dialogues sont une des briques de base pour la programmation de chatbots par composition d'autres chatbots. Cela correspond à de l'appel de fonction. Enfin, la fonction `Wcs_bot.interpret` interprète la variable de contexte `skip_user_input` qui active plusieurs transitions de l'automate de dialogue sans demander de nouvelles entrées à l'utilisateur. La signature complète de cette fonction est la suivante :

```
val interpret :
  ?before:(message_request -> message_request) ->
  ?after:(message_response -> message_response) ->
  credential -> string -> message_request -> string * message_response * json option
```

En plus de la réponse et de la valeur de `return` extraite du contexte, cette fonction renvoie l'identifiant du workspace à exécuter lors du prochain tour de conversation (l'identifiant peut changer quand on appelle un sous-dialogue). Les arguments optionnels `~before` et `~after` permettent d'exécuter des fonctions arbitraires avant et après l'appel à `message` comme illustré sur la figure 1.

Programmer l'application La fonction `Wcs_bot.interpret` n'exécute qu'un seul tour de conversation. Mais nous pouvons maintenant définir simplement un chatbot qui traduit la blague à la volée pour un utilisateur francophone :

```
let exec creds ws_knockknock =
  let rec loop ws_id ctx =
    let txt = input_line stdin in
    let req = Wcs.message_request ~text: txt ~context: ctx () in
    let ws_id, rsp, return =
      Wcs_bot.interpret ~before: englify ~after: frenchify creds ws_id req
    in
    let () = List.iter print_endline rsp.msg_rsp_output.out_text in
    begin match return with
    | Some v -> v
    | None -> loop ws_id rsp.msg_rsp_context
    end
  in
  loop ws_knockknock `Null
```

Les fonctions `englify` et `frenchify` permettent de passer de l'anglais au français en utilisant par exemple l'API Watson Language Translator [11], ou l'API Google Translate [7]. Notre blague devient alors :

> *Bot: Toc Toc*
> *User: Qui est là ?*
> *Bot: Crayon Brisé*
> *User: Crayon Brisé qui ?*
> *Bot: Peu importe c'est inutile...*

Déployer un chatbot La bibliothèque `wcs-ocaml` fournit aussi les fonctions pour déployer les workspaces sur le service WCS. Il est ainsi possible d'intégrer le déploiement des workspaces dans l'application. Cela évite des problèmes d'incompatibilité de versions entre les deux parties du système. Faciliter le déploiement de l'application permet des mises à jour fréquentes. Les programmeurs peuvent ainsi prendre rapidement en compte les retours d'expériences des utilisateurs pour améliorer le chatbot [22].

4 État de l'art

Les systèmes de dialogue permettent le développement d'interfaces interactives basées sur le langage naturel [1]. La popularités des assistants personnels intelligents (tels que Siri ou Alexa) contribue à l'intérêt croissant dans ce domaine. McTear [17] et Jurafsky et Martin [13, Chapitre 29] donnent un aperçu des différentes technologies disponibles. L'utilisation d'automates finis est un modèle relativement expressif mais assez bas-niveau [8, 22]. Les *frames* sont une alternative spécialisée pour remplir des formulaires [2, 15]. Les systèmes plus récents [20] combinent un ou plusieurs services (par exemple pour obtenir un interprète de langage naturel plus performant) avec une bibliothèque intégrée dans un langage de programmation [18, 5].

La notion de workspace dans Watson Conversation System fournit un modèle attrayant, car plus déclaratif et offrant une interface de visualisation. L'un des inconvenient de cette approche est la nécessité de développer les chatbots en deux parties : le workspace et l'application correspondante. L'utilisation de `wcs-ocaml` fournit une solution complète et répond à ce problème. Plusieurs autres SDK pour WCS sont disponibles [8] (entre autres pour Python, Node et Java). `wcs-ocaml` est le premier SDK pour WCS qui intègre la construction du workspace et de l'application et qui fournit un model pour l'appel de sous-dialogue. Il existe également d'autres services de chatbot associés à leur propre SDK [6, 19].

5 Conclusion

Nous avons présenté `wcs-ocaml`, une bibliothèque *open source* qui permet de programmer des chabots en OCaml avec *Watson Conversation Service*. La bibliothèque `wcs-ocaml` a été utilisée pour développer ChessBot (un chatbot pour ChessEye[2]) et RuleBot (un chatbot pour l'édition de *règles métiers* [4]). Elle a également été utilisée conjointement avec la bibliothèque d'analyse statique *T.J. Watson Libraries for Analysis* (WALA)[3] pour le déboggage de workspaces WCS.

Pour le développement de ChessBot et RuleBot, nous avons dû utiliser des formes de compositions plus avancées que les sous-dialogues: pipeline, parallélisme et préemption. Ces composition s'expriment naturellement dans un langage réactif de haut niveau tel que ReactiveML [16]. La bibliothèque `wcs-ocaml` a permis une intégration rapide en ReactiveML, ce langage étant lui aussi fondé sur OCaml.

RuleBot repose également sur un mécanisme d'échange de données constant (des fragments de l'arbre de syntaxe du langage de règles) entre l'application et le service WCS. L'utilisation de `wcs-ocaml` pour programmer l'application et le workspace nous a permis d'assurer que la construction des ces fragments d'arbre de syntaxe (qui sérialise un type OCaml en structure JSON) et l'interprétation des réponses du moteur de conversation (qui reconstruit un type OCaml à partir de la structure en JSON) sont cohérents.

Remerciements Nous tenons à remercier Avraham Shinnar pour son travail sur le SDK typescript qui a été une source d'inspiration pour `wcs-ocaml`. Nous remercions également Timothy Bourke pour le package LaTeXchecklistings [3] qui nous a permis de vérifier le code présenté dans cet article.

[2] https://github.com/chesseye/chesseye
[3] http://wala.sourceforge.net/wiki/index.php

References

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[2] D. G. Bobrow, R. M. Kaplan, M. Kay, D. A. Norman, H. Thompson, and T. Winograd. GUS, a frame-driven dialog system. *Artificial Intelligence*, 8(2):155–173, 1977.

[3] T. Bourke and M. Pouzet. *The checklistings package*, 2015.

[4] M. J. Boyer and H. Mili. Agile business rule development. In *Agile Business Rule Development*, pages 49–71. Springer, 2011.

[5] Facebook. Messenger platform, 2017. https://developers.facebook.com/docs/messenger-platform/ (Retrieved October 2017).

[6] Facebook. Messenger platform, 2017. https://developers.facebook.com/docs/messenger-platform/ (Retrieved December 2017).

[7] Google. Google translate service, 2017. https://cloud.google.com/translate/ (Retrieved October 2017).

[8] IBM. *Overview of the IBM Watson Conversation service*, 2017. https://www.ibm.com/watson/developercloud/doc/conversation/index.html.

[9] IBM. Watson conversation service api, 2017. https://www.ibm.com/watson/developercloud/conversation/api/v1 (Retrieved October 2017).

[10] IBM. Watson conversation service documentation, 2017. https://console.bluemix.net/docs/services/conversation/expression-language.html (Retrieved October 2017).

[11] IBM. Watson language translator service, 2017. https://www.ibm.com/watson/services/language-translator/ (Retrieved October 2017).

[12] M. Jambon. *atdgen documentation*, 2017. https://mjambon.github.io/atdgen-doc (Retrieved October 2017).

[13] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall, second edition, 2009.

[14] R. Kaplan. Beyond the GUI: It's time for a conversational user interface. *Wired*, 2013.

[15] B. Lucas. VoiceXML for web-based distributed conversational applications. *Communications of the ACM (CACM)*, 43(9):53–57, 2000.

[16] L. Mandel, C. Pasteur, and M. Pouzet. ReactiveML, ten years later. In *Proceedings of 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, 2015.

[17] M. F. McTear. Spoken dialogue technology: Enabling the conversational interface. *ACM Computing Surveys (CSUR)*, 34(1):90–169, 2002.

[18] Microsoft. Bot framework documentation, 2017. https://docs.microsoft.com/en-us/bot-framework/ (Retrieved October 2017).

[19] Microsoft. Bot framework documentation, 2017. https://docs.microsoft.com/en-us/bot-framework/ (Retrieved December 2017).

[20] A. Patil, K. Marimuthu, R. Niranchana, et al. Comparative study of cloud platforms to develop a chatbot. *International Journal of Engineering & Technology*, 6(3):57–61, 2017.

[21] Spring. Spring expression language (SpEL), 2017. https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#expressions (Retrieved October 2017).

[22] J. D. Williams, N. B. Niraula, P. Dasigi, A. Lakshmiratan, C. G. J. Suarez, M. Reddy, and G. Zweig. Rapidly scaling dialog systems with interactive learning. In *International Workshop on Spoken Dialog Systems*, 2015.