# Programming Language Support for Natural Language Interaction

Alex Renda
Cornell University

Harrison Goldstein
Cornell University

Sarah Bird
Facebook

Chris Quirk
Microsoft Research

Adrian Sampson
Cornell University

## ABSTRACT

Modern conversational user interfaces depend on natural language understanding (NLU) engines, but integrating these capabilities creates a new category of engineering challenges. Developers write verbose, unsafe code to intermediate between NLU services and application logic, and ambiguous parses further complicate handling. We present a DSL for configuring an NLU model that ensures consistency and type safety, and we propose a new language construct to express ambiguity by exploring hypothetical actions.

## 1 INTRODUCTION

Advances in natural language understanding (NLU) have engendered a new generation of chatbots, voice-based intelligent assistants, and conversational interfaces [3, 4, 6]. Excitement over language-based interactions, however, has run ahead of engineering tools to support them. We propose new language constructs to mitigate sources of complexity when designing NLU applications.

First, current NLU tools require error-prone boilerplate code. The root problem is that they force programmers to specify a language model twice: once in a special configuration interface, and once in the code that processes the results from this configuration. We propose a *type-directed* approach to specifying the language model, where a single set of type declarations completely determines how utterances are translated into structured data.

Second, natural language is ambiguous. Applications need to provide domain evidence to help resolve user intent and choose the best action among alternatives. We propose a new language construct, *hypothetical worlds*, that lets programmers explore a space of alternative interpretations while controlling their effects.

## 2 TYPE-DIRECTED NLU INTEGRATION

Modern cloud NLU services such as Wit.ai, LUIS, Dialogflow, and Lex[1] make it easy to start building a conversational interface. Consider using Wit.ai to build an interface to a calendaring application that supports scheduling, moving, and listing meetings. We first use Wit.ai's web-based GUI to configure the *entities* that the language model should extract from these utterances. We also provide several example messages for each command and label the entities they contain. For example, we might enter the utterance *set up a Monday meeting with Valeria*, classify its *intent* as `Schedule`, highlight *Valeria* as a `Person` entity, and mark *Monday* as a `Day` entity.

The next step is to add the client code to the application. The code needs to invoke Wit.ai on an utterance and parse the response to dispatch to application logic. The goal for the `Schedule` intent might be to call a function `schedule(who, when)`. Ideally, Wit.ai would translate the message *ask Wei to meet Tuesday* to a JSON structure that directly corresponds to the `schedule` function's signature:

```
{ tag: 'Schedule',
  data: { who: 'Wei', when: { day: 'Tuesday' } } }
```

The client code could use a `switch` statement to dispatch to handler functions and supply their arguments from `data`. The reality of natural language makes the real response format more complex:

```
{ Intent: ['Schedule'], Person: ['Wei'], Day: ['Tuesday'] }
```

Wit.ai and similar services specify the tags for the entities, but they do not specify how to map the entities onto function arguments. They typically return a list of possibilities for each entity. Worse, the lists may be empty: with insufficient training, the `Person` entity might be missing altogether. The application client code has to handle all of these possibilities to construct the correct function call. Because this glue code is nontrivial, it can easily get out of sync with the NLU model: the language's type system cannot catch a misspelling in the name of an entity, for example.

**A type DSL for NLU models.** We propose a domain-specific language that eliminates this boilerplate by jointly specifying the NLU configuration and the data types for its results. Developers can skip the configuration GUI and write equivalent declarations in our DSL:

```
free-text Person; free-text Time;
keywords Day = "Sunday" | "Monday" | ... | "Saturday";
alias Date = { day: Day, time?: Time };
trait Intent =
  | <Schedule> { who: Person, when: Date }
  | <Move> { from: Date, to: Date }
  | <List> {};
```

Each of the `free-text`, `keywords`, and `trait` keywords declare both a type for application code and a corresponding entity in the NLU model. The three keywords reflect Wit.ai's three *search strategies* for entities. Our compiler [7] translates a DSL program into both type declarations (in TypeScript) and a language model (for Wit.ai). For example, this code declares a new type `Person` for the application, as a type alias for `string`, and a `Person` free-text entity in the NLU model to extract those strings. The `alias` keyword does not define an entity; instead, it abstracts composite structures like `Date`.

**Recovering structure from flat NLU data.** When translating from a Wit.ai API response to a declared type, the primary challenge is recovering nested structure. In simple cases, reconstruction is manageable: the `Schedule` intent, for example, searches for a `Person` entity, a `Day`, and an optional `Time`. Reconstruction becomes more complex when the responses can be ambiguous. Consider a response for a `Move` message, containing two `Day` entities. The NLU engine only knows that the user mentioned two days, but it does

[1] http://wit.ai, http://www.luis.ai, http://dialogflow.com, http://aws.amazon.com/lex

not have enough information to decide which `Day` value to match to the `from` field in the `Move` record and which to use as the `to` field.

Our implementation prohibits this ambiguity by ruling out duplicate entities in each trait branch. This limitation reflects Wit.ai's flat response format, which only tags values with entity kinds. Other NLU services avoid this restriction by assigning entities to named *slots*. In general, we find that viewing NLU toolkits through the lens of types can reveal latent strengths and weaknesses in their expressiveness. In future work, we plan to explore NLU model designs that can support a more expressive language of types.

## 3 HYPOTHETICAL WORLDS

A second challenge in engineering NLU interfaces is the ambiguity of natural language. Even perfect language understanding can yield multiple interpretations. For example, Wit.ai and similar services are unlikely to distinguish between spoken references to Gene and Jean or whether requesting to meet *at eight* refers to 8 AM or 8 PM. Instead, Wit.ai produces multiple, weighted interpretations and leaves the choice to the application. The examples above ignore this uncertainty, but with help from the programming language, applications can exploit it by combining it with domain knowledge.

We propose *hypothetical worlds*, a language construct that expresses nondeterministic choice to search for the best choice among a set of possible actions. In a hypothetical world, the execution buffers effects temporarily until the program *commits* its changes. Programmers can write in a natural style, as if the code were interacting with the real world, but only commit to changes based on a potential action's outcome. There are two fundamental operations: a `hyp` statement runs a block of code in a hypothetical context and returns a *world* value, and a `commit` operation atomically applies the effects from a world value to the currently-executing world. Consider the following example, where the assignment to `x` is hypothetical until `world` is `commit`ted:

```
x := 1;
world := hyp {
  x := 2;
  assert x == 2;  // Local effects are visible.
}
assert x == 1;    // The world has not yet been committed.
commit world;
assert x == 2;    // The world's changes have been merged.
```

While `hyp` is useful on its own, we also use it to build higher-level constructs for dealing with uncertainty. For example, `search` and `max` let programs explore a set of possible alternatives, such as the results of an NLU model. The following code parses an utterance and hypothetically executes each possible interpretation:

```
interps := parse(utterance);
worlds := search (likelihood, name, time) in interps {
  person := contacts.get(name);
  calendar.schedule_meeting(person, time);
  fitness := calendar.fitness() * likelihood;
}
commit worlds.max(fitness);
```

This example commits the changes from the interpretation that maximize a fitness metric on the updated calendar. For example, `fitness` might reward scheduling the meeting back-to-back with other appointments to preserve larger free blocks for work.

**Semantics of hypothetical execution.** The semantics for `hyp` are based on transactions and fork–join parallel programming [2]. At the beginning of a `hyp` block, the execution forks the program state. Updates in each world are isolated from each other, with two exceptions: a parent world can read *weight* values from its children, like `fitness` above, and it may `commit` a child to merge its changes into the parent world's state. Programs can nest hypothetical worlds arbitrarily deeply. Side effects that change the environment, such as I/O, only take effect in the top-level world.

As in any system with partially-ordered access to shared state, conflicting updates are possible. Our language has no explicit mechanism for resolving conflicts: instead, the merge fails, as in an aborted transaction, and the program can explore an alternative. In the case of `search`, this may involve falling back to the next-best world, and attempting to merge that, and iterating until either a feasible world is found or the entire parameter space is exhausted.

To formalize this model, we give a sampling of our operational semantics [7] for a language with hypothetical worlds. In this semantics, $\sigma$ is a variable store, $\omega$ stores the state of child hypothetical worlds, $\mu$ is a (partial) merge function, and $c$ is a command:

$$\frac{\langle c, \sigma_{\text{orig}}, \varnothing, \mu\rangle \Downarrow \langle \sigma_{\text{hyp}}, \omega_{\text{hyp}}\rangle}{\langle u := \texttt{hyp } \{\ c\ \}, \sigma_{\text{orig}}, \omega, \mu\rangle \Downarrow \langle \sigma_{\text{orig}}, \omega[u \mapsto \sigma_{\text{hyp}}]\rangle}$$

This rule, for `hyp`, executes `c` in the original state $\sigma_{\text{orig}}$ and an empty $\omega$ to produce a final state $\sigma_{\text{hyp}}$, which is stored in the original $\omega$.

$$\frac{\omega[u] = \sigma_{\text{hyp}} \qquad \begin{array}{c}\forall v \notin \sigma_{\text{hyp}}.v \notin \sigma_{\text{merge}} \\ \forall v \in \sigma_{\text{hyp}}, \sigma_{\text{merge}}[v] = \mu(\sigma_{\text{curr}}[v], \sigma_{\text{hyp}}[v])\end{array}}{\langle \texttt{commit } u, \sigma_{\text{curr}}, \omega, \mu\rangle \Downarrow \langle \sigma_{\text{merge}}; \sigma_{\text{curr}}, \omega\rangle}$$

The `commit` rule reads $\sigma_{\text{hyp}}$ from $\omega$, attempts to merge any changes between it and $\sigma_{\text{curr}}$, and concatenates the merged state to $\sigma_{\text{curr}}$.

**Distributed hypothetical execution.** Like a transaction, a hypothetical world publishes its effects atomically in its parent world when it commits. This atomicity extends to interactions with the outside world: hypothetical worlds can help synchronize accesses to external systems and aid coordination among multiple users. For example, a distributed calendaring system can use input from multiple users' calendars to arrive at a collective meeting time. To commit the addition to all calendars simultaneously, the runtime uses a distributed commit protocol to ensure that no user has added a conflicting event since the process began. The system can synchronize external services by layering a concurrency control shim between it and the program with hypothetical worlds.

## 4 OPEN QUESTIONS AND OPPORTUNITIES

Language-based interaction engineering poses a landscape of challenges beyond the ones discussed here. Collaborative assistants need support for secure cooperation between mutually distrustful users [5]. To resolve more vexing sources of ambiguity, agents need to iteratively refine queries through multi-turn interaction, so languages need to support efficient incremental updates [1]. Complex ecosystems need compositionality: developers should be able to build small behaviors within module abstractions and combine them to build up larger interaction designs. Each challenge recalls classic ideas from programming languages that are ripe for application in this new domain.

# REFERENCES


[1] Umut A. Acar. 2005. *Self-adjusting Computation.* Ph.D. Dissertation. Carnegie Mellon University.

[2] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent Programming with Revisions and Isolation Types. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[3] Hamza Harkous, Kassem Fawaz, Kang G. Shin, and Karl Aberer. 2016. PriBots: Conversational Privacy with Chatbots. In *Symposium on Usable Privacy and Security (SOUPS).*

[4] IBM. [n. d.]. Watson Health. https://www.ibm.com/watson/health/

[5] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. [n. d.]. Fabric: a platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP).*

[6] Microsoft. [n. d.]. Bot Framework. https://dev.botframework.com

[7] Alex Renda, Harrison Goldstein, Sarah Bird, Chris Quirk, and Adrian Sampson. [n. d.]. Opal source and semantics. https://capra.cs.cornell.edu/research/opal/