Typed Interactions for NLU in Opal

Harrison Goldstein

Fall 2017

Abstract

Natural Language Understanding (NLU) is a powerful tool for modern applications. Interacting with remote NLU engines can be error prone: most backends provide no type guarantees, and configuration synchronization is a constant source of bugs. We present a DSL for configuring a NLU application that ensures synchronization and type-safety.

I. Introduction

In the 1970's, a text-based command interface was "good enough" for people interacting with computers. After all, most of the computation being done was fairly specific to some application domain, the person operating the computer was well-trained, and the machine was viewed as a tool for getting the job done.

Today, computers are our personal assistants. They are integrated with our lives and they help everyday people do everyday tasks. Most people who use computers today never touch a command shell, and even those who do would often rather not. Graphical interfaces are amazingly powerful, and are often sufficient to allow users to easily interact with the computer, but even those have their drawbacks. We would really like our personal assistant to respond to commands in human language:

- "Hey Siri! Play some music."
- "Ok Google. Navigate to my hotel."
- "Slackbot, set my status to active."

This is the realm of Natural Language Understanding (NLU).[1]

[1]We do not distinguish between spoken or written language when talking about NLU in general.

We have seen a huge growth in NLU applications in recent years, and we will likely see more. Unfortunately, there has been no serious effort made to make this power available at the language level, so programmers have been forced to integrate these technologies into their applications by hand. We propose an addition to the Opal language that will begin to close this gap. [5]

a. An Example

Consider a simple *reminders* interface to a calendar or other scheduling assistant. A user might ask the application to set a reminder containing a message and a time (for simplicity, we will just allow the user to specify `later` or `tomorrow`) or she might ask the application to get any active reminders. Since Opal is embedded in TypeScript, [4] a typical Opal application might express this interface with the following declarations:

```
type Msg = string;
type Time = "later" | "tomorrow";
type Set = {
  msg: Msg,
  time: Time
};
type Intent =
  | { tag: "Set", data: Set }
  | { tag: "Get", data: {} };
```

We would like to interact with this application via human language, so we need some way of converting a user's *utterance* to something of type `Intent`. Figure 1 shows the desired data representation of a user utterance.

```
{
  tag: "Set",
  data: {
    msg: "pick up the dry cleaning",
    time: "later",
  }
}
```

Figure 1: The desired data representation of the utterance: *Remind me later to pick up the dry cleaning.*

b. Types for NLU

If a programmer simply wrote down types like the ones above, set up their NLU engine, and started their application, there would be a problem. Since there is no connection between the types and the NLU configuration, these types could actually be unsound! If the programmer made a mistake in *either* the types *or* the configuration, the application would be unusable and no compiler could possibly know. Bugs like these are particularly insidious because they might be mistaken as network errors or even just as an indicator of under-training.

In this work, we set out to connect the types and the NLU configuration, allowing the programmer to avoid these awful bugs. There are a few ways that one might try to do this. First, the programmer could write down the NLU configuration and extract the types from it. This is pretty much a non-starter, since there is fundamentally less information in the NLU configuration than there is in the type definitions. Instead, the programmer could write the types, and extract the NLU information from those. This approach is more feasible, but it would require some kinds of annotations in the types to capture implementation-specific properties of the configuration.

We present a third option: a domain specific language (DSL) for simultaneously specifying

- configuration for an NLU engine;
- a typed interface for responses

This allows the user to tune the configuration as much as they need, to write useful types, and to ensure that these two artifacts are always compatible.

We will begin by presenting a technical description of our type language in Section II. In Section III, we present a formal specification for the language our type-directed response parsing algorithm. Section IV covers the implementation of the language and related libraries. Finally, Sections V and VI present an evaluation of our work, and our plans for future work, respectively.

II. Language Description

For our initial work, we elected to focus on building a system to support a single NLU backend—specifically, we are working with an NLU engine called Wit. [3] When configuring Wit, as with many NLU backends, the main task is setting up the *entities* that Wit should be looking for. An entity is a discrete piece of information that the NLU engine extracts from an utterance. In our example, the messages and the time are both entites, as is the user's intent.

a. Search Strategies

When entities are declared, the user must specify a "search strategy." In the case of Wit, there are three main strategies:

Free Text Entities that are found via free text matching are just strings of text from the utterance. In our example, the message is a free text entity.

Keywords Keywords entities are just words from some pre-defined set. Things like `later` and `tomorrow` would be captured via keyword searching.

Trait These are slightly more abstract. Traits capture something that is a quality of the utterance as a whole. Something like a user's intent or even their mood could be a trait.

To some, these lookup strategies might seem like an implementation detail, but there is actually something important going on here. The search strategies are closely tied to the *type* of the desired output. In the case of free text and keywords, this association is obvious: free

```
free-text Msg
keywords Time = "later" | "tomorrow"
alias Set = {
  msg: Msg,
  time: Time
}
trait Intent =
  | <Set> Set
  | <Get> {}
```

Figure 2: Some DSL declarations.

text entities are just `string`s, and keywords are members of a union of literal types.[2] Traits are slightly more complex, but they essentially represent a tagged union, where the extracted entity itself carries only the tag. We will make these ideas more concrete in section III.

b. The DSL

With our observations about search strategies in mind, we can write our types from before in a slightly different way. Figure 2 shows the same type declarations from Section I written in our DSL.

The main thing to notice is that the keyword `type` has been replaced by entity declarations `free-text`, `keywords`, and `trait`, or by `alias`. There are a number of benefits of entity declarations. First and foremost, it allows the user to explicitly define the NLU engine configuration alongside the program types. With types defined in our DSL, we can automatically generate configuration for entities that is guaranteed to be consistent with the types in the program. When a `Time` is returned to the user, it will be guaranteed to be either `"later"` or `"tomorrow"`. This alone solves a lot of the problems that tend to come up when writing these types of applications.

Another benefit of these bespoke declaration forms is simplified syntax. When a `free-text` entity is declared, there is no need to write "`= string`," because that is implied by the search strategy. Similarly, since traits are always tagged unions, we can write them in (almost) ML syntax, rather than writing out the standard TypeScript workaround.

[2]For example, `"later" | "tomorrow"`.

$$
\begin{aligned}
\mathbb{D} ::= &\ \mathrm{FreeText}(\mathsf{t}) \\
&\mid \mathrm{Keywords}(\mathsf{t},\ [\ell]) \\
&\mid \mathrm{Trait}(\mathsf{t},\ \{s : \mathbb{T}\}) \\
&\mid \mathrm{Alias}(\mathsf{t},\ \mathbb{T})
\end{aligned}
$$

$$
\mathbb{T} ::= \mathrm{Def}(\mathsf{t}) \mid \ell \mid \{s : \mathbb{T}\}
$$

$$
\mathsf{t} \in \mathsf{Tag} \qquad \ell, s \in \textbf{string}
$$

Figure 3: The abstract syntax tree for our DSL.

Finally, writing specific entity declarations rather than just `type`, makes it clear that there is more going on than a direct translation into TypeScript. In fact, the DSL parser does lot of validation to guarantee that only reasonable types can be written down. For example, a `keywords` declaration is required to be a union, and that union must contain only literals. Even the `alias` keyword has some validation built in. Some TypeScript types would be problematic if they were allowed—we enforce those conditions in the DSL.

Figure 3 shows the AST that is produced when we parse the DSL. Looking at this, it also becomes clear that the limitations on the types allowed in a trait declaration are fairly strict. As of writing, there are only three types allowed within a trait declaration: previously defined declarations, literals, and records (where each element is subject to the same restrictions). These limitations match up with the capabilities of NLU engines in general. While many backends provide the ability to parse numbers, dates, and other data, these types of entities are implementation specific. In Section VI, we discuss plans to expand the allowed types in a modular way.

c. NLU Engine Responses

Our DSL solves quite a few problems on its own. It forces the program types to agree with the NLU configuration, and it provides a convenient way to write both down together. It also prevents the user from having unreason-

```
{
  Intent: ["Set"],
  Msg: ["pick up the dry cleaning"],
  Time: ["later"]
}
```

Figure 4: The unstructured response for the utterance: *Remind me later to pick up the dry cleaning.*

able expectations of the backend, by restricting the types that a user can expect as a response.

Unfortunately, there is still one major issue. The vast majority of NLU engines do not return structured data. Instead, they return a bag of entities with no information about the relationships between them. Figure 4 shows a sample response. Recall that in our example, `Msg` and `Time` entities were only expected when there was an `Intent` with tag `"Set"`. Furthermore, our types dictate exactly where in the data structure those entity values should end up. The popular libraries, Wit incuded, igore this structure entirely.

This makes it necessary to have a run-time component of the system that knows how to reconstruct structured data from the unstructured response that comes from the NLU engine. Building structured data from the response is a nontrivial task. First, it requires that information about the relationships between entities and data be available at runtime. This is mostly an engineering problem, and our solution is outlined in Section IV.

The other difficulty is that the types might admit more than one valid response object. To see how this works, imagine that we want to allow a user to set two messages at once. We would change the `Set` type to be:

```
type Set = {
  msg1: Msg,
  msg2: Msg,
  time: Time
};
```

Unfortunately, this kind of ambiguity makes it impossible to find a principled way of parsing a response. Since the backends do not expose ordering information, a response with two `Msg`

$$\begin{aligned}\mathcal{D} ::= &\ \text{FreeText}(\mathsf{t}) \\ \mid &\ \text{Keywords}(\mathsf{t},\ [\ell]) \\ \mid &\ \text{Trait}(\mathsf{t},\ \{s : \mathcal{T}\})\end{aligned}$$

$$\mathcal{T} ::= \mathcal{D} \mid \ell \mid \{s : \mathcal{T}\}$$

$$\mathsf{t} \in \mathsf{Tag} \qquad \ell, s \in \textbf{string}$$

(a) The core language.

$$\tau ::= \ \textbf{string} \mid \ell \mid \{s : \tau\} \mid \langle \tau \rangle$$

(b) Opal types.

Figure 5: Formal structures.

fields might parse to any of four different objects. We say four because the reconstruction algorithm might reasonably ignore one of the messages and just put the same one for `msg1` and `msg2`. For this reason, we impose the restriction that declarations do not reuse entities.

III. Formalism

In order to reason about the correctness of our translations, we have formalized the results from Section II. We begin by writing down a core language for the type DSL as shown in Figure 5a. There are similarities between these definitions and the ones in Figure 3, but the core language has some slightly different goals. First, we remove aliases and require that all type definitions are written "in-line." Next, rather than split the definition by allowing tags within $\mathcal{T}$, we also require that entity definitions are inlined. This means that the data a user expects from the NLU backend can be represented by a single tree of type $\mathcal{D}$. Importantly, none of these simplifications are fundamental—this formulation can represent the same types as the actual system. This approach simply gives up some usability to simplify reasoning.

In addition to a grammar for the core language itself, in Figure 5b we define a grammar of types, τ. We can think of τ as representing

$$\begin{aligned}
&\mathsf{interp}(\text{FreeText}(\mathsf{t})) = \textbf{string} \\
&\mathsf{interp}(\text{Keywords}(\mathsf{t}, [\ell])) = \langle \ell \rangle \\
&\mathsf{interp}(\text{Trait}(\mathsf{t'}, \{s : t\})) = \\
&\qquad \langle \{\text{tag} : s,\ \text{data} : \mathsf{interpTy(t)}\} \rangle \\
\\
&\mathsf{interpTy}(d) = \mathsf{interp(d)} \\
&\mathsf{interpTy}(\ell) = \ell \\
&\mathsf{interpTy}(\{s : t\}) = \{s : \mathsf{interpTy}(t)\}
\end{aligned}$$

Figure 6: Type interpretation.

the final form of the data that a user hopes to obtain, with no information about the entities that generated the data. They map almost directly into types that are used by the underlying Opal system. We add a few types that are noteworthy. The type **string** is added to account for FreeText entities. The type $\langle \tau \rangle$ represents an arbitrary union and covers the cases of Keywords and Trait entities.[3]

As mentioned above, user's view of their system can be formalized as a top-level declaration generated by $\mathcal{D}$. Since this declaration encompases both type and configuration information, we would like a way to compute just the *type* of data that the user expects from the NLU engine. We define the following mutually recursive functions, which traverse a declaration and construct a single type in τ.

$$\begin{aligned}
\mathsf{interp} &: \mathcal{D} \to \tau \\
\mathsf{interpTy} &: \mathcal{T} \to \tau
\end{aligned}$$

We implement interp and interpTy in Figure 6.

One interesting thing to point out is that the record in the Trait declarations becomes a union type. This makes sense, since a trait represents a property of the utterance as a whole, and thus can decide the structure of the rest of the response.

With the basic building blocks defined, we can move forward in formalizing our system.

[3]This notation is slightly unconventional, but we prefer to write $\langle \tau \rangle$ or $\langle \tau_1 \mid \cdots \mid \tau_n \rangle$ rather than simply $\tau_1 \mid \cdots \mid \tau_n$ to separate it from the surrounding syntax.

a. Configuration Generation

The first step that our system takes is configuration. In practice, this is a two-part process: the user needs to extract the appropriate Opal code as well as the configuration for the NLU backend. The first step is covered by our definition of interp. Since we are taking τ as an analog for relevant Opal types, the type generation process is exactly the execution of interp on the top-level entity declaration.

As for the second step, we would like the NLU backend to be set up such that its responses are *well-formed* with respect to the configuration. Intuitively, any time we look up some entity tag in the response, we would like to get data that is compatible with that entity's declaration. We define a type, Resp of responses, and require that a lookup function exists, with the following type:

$$\mathsf{lookup} \ : \ \mathsf{Tag} \to \mathsf{Resp} \rightharpoonup \textbf{string}$$

Note that the function is partial; there is no guarantee that the response will contain the desired data. In the following development, we will often assume that lookup is defined on tags we care about, but that assumption will be stated explicitly rather than assumed by default.

The lookup function always returns a string, but the string might mean any number of things. The data might be a particular literal, in the case of a keywords or trait entity, or an entire phrase in the case of a free text entity. We define configurations as functions that map a subset of entity tags to the type of data expected for those entities. The function cfg takes a declaration and produces a configuration for that declaration.

$$\mathsf{cfg} \ : \ \mathcal{D} \to \mathsf{Tag} \rightharpoonup \tau$$

We define cfg in Figure 7. The exact details of the else branch of the Trait case are implementation specific, but the basic idea to map the cfg function over all nested declarations. This achieves the goal of capturing every tag in the declaration.

Now, we call a response well-formed with respect to $d : \mathcal{D}$ if and only if it returns appropriate data on lookup. More formally, we say that

$$\begin{aligned}
&\mathsf{cfg}(\text{FreeText}(\mathsf{t}), \mathsf{t}) = \textbf{string} \\
&\mathsf{cfg}(\text{Keywords}(\mathsf{t}, [\ell]), \mathsf{t}) = \langle \ell \rangle \\
&\mathsf{cfg}(\text{Trait}(\mathsf{t}', \{s : t\}), \mathsf{t}) = \\
&\qquad \textbf{if } \mathsf{t} = \mathsf{t}' \textbf{ then } \langle s \rangle \textbf{ else } \ldots
\end{aligned}$$

Figure 7: Configuation generation.

for some $r : \mathsf{Resp}$,

$$\begin{aligned}
&r \text{ well-formed}_d \iff \\
&\quad \forall(\mathsf{t} : \mathsf{Tag}).\ \mathsf{cfg}(d, \mathsf{t}) \text{ defined} \Rightarrow \\
&\qquad \mathsf{lookup}(\mathsf{t}, r) : \mathsf{cfg}(d, \mathsf{t})
\end{aligned}$$

We will use well-formedness as our basic signal that a the application and the response have been configured to be compatible.

b. Response Parsing

We discussed above that there is a challenge in parsing the unstructured response from the NLU engine into data of the appropriate types. This section will formalize that problem and present our solution.

Our goal will be to write a parsing function which takes a declaration, d, and a response, and produces a value of type $\mathsf{interp}(d)$. We will again write two mutually recursive functions, parse and parseTy, with types:

$$\begin{aligned}
\mathsf{parse} &: \Pi_{(d:\mathcal{D})}.\mathsf{Resp} \rightharpoonup \mathsf{interp}(d) \\
\mathsf{parseTy} &: \Pi_{(t:\mathcal{T})}.\mathsf{Resp} \rightharpoonup \mathsf{interp}(t)
\end{aligned}$$

Since we need the type of the result to depend on the declaration, these are dependent functions. The functions are also partial because lookups might fail.

In the event that r is not well-formed with respect to the configuration generated by d, $\mathsf{parse}(d, r)$ is undefined. Otherwise, we define parse by case analysis on the declaration.

$$\mathsf{parse}(\text{FreeText}(\mathsf{t}), r) = \mathsf{lookup}(\mathsf{t}, r)$$

This case is fairly straightforward. Since the interpretation of a free text entity is just a **string**, we can simply look up the value of the appropriate entity from the response.

$$\mathsf{parse}(\text{Keywords}(\mathsf{t}, [\ell_1, \ldots, \ell_n]), r) = \mathsf{lookup}(\mathsf{t}, r)$$

Perhaps surprisingly, we can use the exact same definition in the Keywords case that we did in the FreeText case. We are required to return a value of type $\langle \ell_1 \mid \cdots \mid \ell_n \rangle$; as long as r is well-formed, it will have been configured to have the same list of literals. We are guaranteed to get one of them back from lookup.

$$\mathsf{parse}(\text{Trait}(\mathsf{t}, rec), r) = \mathsf{parseTy}(rec[\mathsf{lookup}(\mathsf{t}, r)], r)$$

The final case for parse simply matches the value in the response to an index into the record, and interprets that record entry using parseTy. It is important to note that since interp returns a union here, we only need to return a value that is in some branch of that union. In this case, we use the lookup value to decide which branch.

We will not write out the full definition of parseTy, but note that it is fairly automatic. If the type is another declaration, we can call back to parse: if it is a literal we simply generate the appropriate string; and if it is a record, we build the appropriate record and recursively build out its fields.

c. Correctness

We would like to bring both parts of the formalism together to show that our system works the way we would like it to. The main property that we would like to show is that parse is correct.

Theorem 1 (parse Correct)**.** *Let d be a declaration in $\mathcal{D}$. Then,*

$$\begin{aligned}
&\forall\ (r : \mathit{Resp}). \\
&r \ \mathit{well\text{-}formed}_d \Rightarrow \mathsf{parse}(d, r)\ \mathit{defined}
\end{aligned}$$

Proof. By induction on d. □

Note that since parse is a dependent function, it will already return a value of the correct type. Thus, we only need to care that it is defined.

IV. Implementation

A high-level workflow for our implementation is shown in Figure 8. The programmer writes types in our DSL and uses our Haskell tool to parse the declarations. The tool serializes three artifacts: a set of TypeScript type declarations, an AST datastructure, and a set of JSON configuration files for Wit. The programmer simply imports the types and AST into their Opal project, uploads the JSON files to Wit, and the system is ready to go.

a. Configuration Tool

When a user wants to configure an Opal application to interact with an NLU library like Wit, they begin by using our configuration tool. The tool ensures that the NLU engine responses will agree with the type definitions in the user's application.

Our configuration tool is written in Haskell, and is designed to be simple and obviously correct. We use the Parsec library to parse the type DSL into the intermediate representation mentioned in Figure 9. [1]

```
data Decl = FreeText String
          | Keywords String
                     [String]
          | Trait String
                  [(String, Ty)]
          | Alias String Ty
          deriving (Show)

data Ty = Def String
        | Lit String
        | Rec [(String, Ty)]
        deriving (Show)
```

Figure 9: The Haskell code for our AST.

After parsing with Parsec, we apply versions of the interp and cfg functions from Section III to generate the appropriate configuration.

b. Run-time Library

At run-time, a typical application using NLU will go through a fairly consistent cycle.

1. Ask user for input, converting voice to text if necessary.
2. Query the NLU engine and obtain a response for the given input.
3. Parse the response into application data.
4. Process parsed data, and repeat.

Steps 1 and 4 are generally application-specific, so we have elected to provide a library which handles steps 2 and 4. For each backend that Opal supports,[4] we provide utilities for querying the API and parsing the response data. The network code involved in obtaining the response is fairly uninteresting, so we will focus on the parsing.

In section III, we outline the parse function, which takes a declaration and a response and generates an Opal object. The function that we provide in the Opal library is similar, but there are two important differences. First, the parse function takes some element of $\mathcal{D}$, which it uses to deduce the structure of the desired output. In the real application, the analog of $\mathcal{D}$ is the `Decl` type in the Haskell code; this is not available to the Opal application at run time.

We get around this by using a modified version the a TypeScript library called Runtypes. [2] Runtypes allows us to to serialize declarations directly into TypeScript and therefore to keep necessary information around at runtime. We extend Runtypes to include a type for entities—which map to declarations in $\mathcal{D}$. Runtypes provides the infrastructure for non-entity types in $\mathcal{T}$.

The other difference arises because Opal does not have dependent types. We cannot completely faithfully transcribe the parse function from above, because it is not obvious that it can be typed at all. The solution here is to combine Opal's dynamic typing with Runtypes' dynamic checking feature to create a function that appears to be dependently typed. First, we use dynamic types within the parse. While building a response object, we assume, where necessary, that it has type `any`. Then, once parsing is complete, we use the Runtypes object to check the parsed result, and we cast the result to the desired type. The definition is shown in Figure 10. The cast is safe, so long as the user re-

[4] At the time of writing, Wit is the only supported backend.

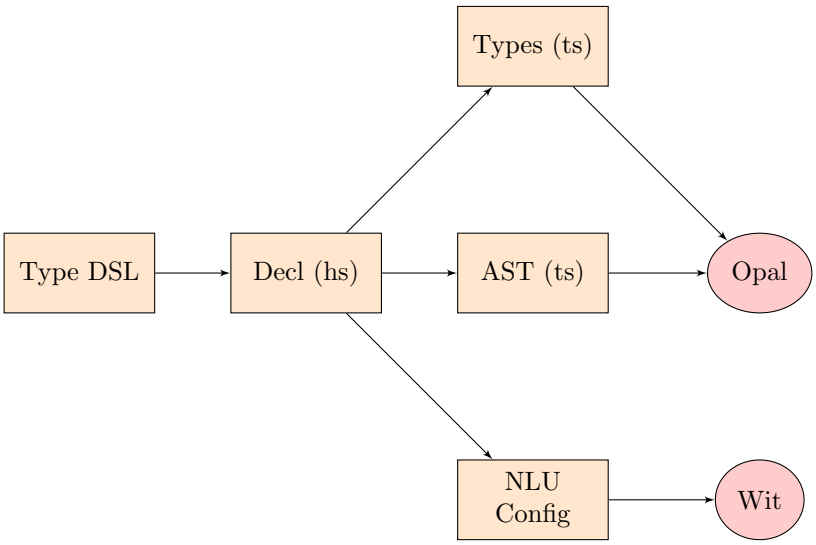

Figure 8: The configuration process.

spects the invariant that parse is always called with compatible type and runtype parameters.

V. Evaluation

Working with the system to build some toy applications led us to a number of important conclusions about its effectiveness. We discuss a number of these conclusions here.

a. Type Expressiveness

The first important thing we found when starting to write types in our DSL is that the standard three entity types are not quite expressive enough for real applications. For example, when trying to implement a small chatbot for scheduling calendar meetings, we wanted to write something like,

```
free-text Person
...
trait Intent = <Schedule> {
  person: Person,
  date: Date,
  time: Time
}
```

but we found that we had no way to express a date or a time. Technically speaking, it might have been possible to encode dates and times as free text and do a second parsing pass in the application code, but that solution is far from elegant. We discuss one potential solution to this problem in Section a.1.

b. Programmer Freedom

When working around unsupported types, there is an obvious tendency to modify the application types to better fit the capabilities of the system. It turns out that this occurs in more cases than just using dates and times. For example, consider a different set of types for calendar scheduling.

```
keywords WkDay = "Sunday" | ...
alias Abs = {
  day: WkDay
  person: Person
}
keywords RDay = "today" | "tomorrow"
alias Rel = {
  day: RDay
  person: Person
}
trait Intent =
  | <Absolute> Abs
  | <Relative> Rel
```

Here, we've done away with the complications of dates and times, and we just allow a user to

```
public parse<T>(rt: Runtype<any>, res: Response): T {
    return rt.check(this._parse(rt, res)) as T;
}

private _parse(rt: Runtype<any>, res: Response): any { ... }
```

Figure 10: The wit parse functions.

specify a day of the week or a "relative day." The interesting observation here is that this is likely not the way that most TypeScript/Opal developers would write that type. For example, one reasonable alternative might express days as a top-level disjunction, and then make the type of intents a normal record.

```
type Day =
  | { tag: "today" }
  | { tag: "tomorrow" }
  | { tag: "abs", day: WkDay }
type Intent =
  { day: Day, person: Person }
```

This makes sense, since `Absolute` and `Relative` are not actully different *intents*, per se. We can see that the system has led us to a different type than we would normally write.

It turns out that this might actually be a good thing, if viewed from the perspective of the NLU engine itself. While `Absolute` and `Relative` are the same intent from a philosophical perspective, they might be parsed very differently. The utterance

Schedule my Wednesday meeting with Alice.

is very different in structure from

Set a meeting with Bob tomorrow.

Thus, it might be the case that forcing the programmer to write slightly different types will make it much easier for the NLU backend to find the correct information. There is likely some work to be done in terms of making our types more idiomatic, but for now we err on the side of making life easier for the NLU engine.

c. Relative Success

There is still a lot to be done on this work and in this space, but as of writing, we feel that we have proven the concept fairly well. The system does the job it intends to, and is usable for actual applications. In general, using the tools feels fairly natural. The Haskell tool produces the output that a user would expect, and the programming interface provides the functionality that a user needs to actually write an application that uses NLU. Overall, we are happy with our approach to the stated problem as well as with the results of the implementation.

VI. Future Work

Moving forward, we hope to take this work in two main directions—wider support for NLU features and backends and entirely new features.

a. Wider Support

Expanding support for more entity types and backends is an obvious way to increase the usefulness of our work. Here, we discuss both of those extensions in detail.

a.1 More Types

In addition to the three "universal" kinds of entities (free text, keywords, and traits), the Wit NLU engine provides more specialized ways of looking up entities. Wit provides support for parsing numbers, dates, times, and other specific forms of text. We elected to keep things simple at first and exclude these built-in entities, but adding support for them seems like a very reasonable next-step.

At a first approximation, this would require modifying the definitions of $\mathcal{D}$ and τ to include declarations for built-in entities and base types for the data associated with those entities. More work needs to be done exploring this

space before any more specific conclusions can be drawn.

a.2 More Backends

We are also interested exploring NLU engines beyond Wit. One reason for this is purely practical—users might want to use other backends, and it would be nice for Opal to support this. Another reason, however, is more academic. Exploring other NLU engines would give us opportunities to test the generality of our current design. Preliminary research suggests that building our type system around entities and search strategies makes sense, but we can only know that for sure if we try to build interfaces for more engines.

b. Interesting Features

There are a number of novel research directions available as extensions to this project. The following are two of the many ideas that we have had.

b.1 Confidence-Based Parsing

Up to this point, we have assumed very little about the data returned in the NLU response. In practice, however, many NLU engines provide more than just information about the matched entities. Wit, in particular, provides confidence levels for each entity that it finds. Thus, a value like `"Set"` might be paired with a confidence level of 70%, meaning that Wit is 70% sure that the user's intent is actually to set a new reminder.

As of now, we do not actually use this information. In the majority of cases that we have tested, a Wit bot either has enough information to get the right entities, or it does not. However, it is possible that as we attempt to support new types of entities and as user types get more complicated, it will be useful to try multiple parses of a single response. In that case, we can choose between different parses using confidence scores.

Hypothetical Worlds It turns out that this process of choosing the parse with the best confidence score is a great application of one of Opal's major features—hypothetical worlds. In a perfect world, we would allow our parser to generate all possible response objects, compute some function based on the confidence values for the entities in the objecs, and choose the object with the best value of the function. Writing this code out by hand would be tedious, inefficient, and error-prone. Hypothetical worlds give this semantics as a language-level feature. When code is executed in a `hyp` block, its computation is entirely hypothetical. Multiple `hyp` blocks can be run in parallel, and their computations can be observed at specified points. Once a particular world is chosen (in this case, by observing the confidence function), a sigle world can be "committed." From that point forward, only the committed world is treated as real, and the computation can continue with the chosen response object.

In the future, we think that hypothetical worlds will open up new opportunities for NLU interaction, including the ability to parse responses into more complex types.

b.2 Training with Tests

A final extension of this project would attempt to close the code/configuration gap even further by allowing users to train their NLU application in a way that is guaranteed to support their tests. We would accomplish this using the same idea behind our type DSL—we introduce a new DSL (or an extension to the current one) for generating both Wit training examples and application tests. While synchronization is less of an issue with training than it is with configuration, it would still be extremely convenient to express the entirety of the application's behavior in one place.

Exploring this path would require a fair amount of implementation work, but we would also need to take great care to avoid overfitting. It is unclear to what extent this would become an issue.

VII. Conclusion

The Opal langauge hopes to be the ideal language for building modern applications that leverage machine learning techniques. As NLU becomes more and more popular as a user interface, it makes sense that Opal would provide

a clean and safe interface for natural langauge interaction. This work contributes a real, working tool in the Opal ecosystem that provides a first attempt at such an interface.

As we move towards a future where the majority of our interactions with computers happen via human language, projects like this will become more and more important. It may be that the approach put forth in the paper paves the way for safe, correct applications that work with NLU. However, even if a different approach wins out, it is almost certain that safe interfaces to NLU engines will eventually become wide-spread. Just as companies have tended to favor strongly typed language for applications where correctness matters, programmers working with NLU will eventually see the need to do so in a principled way.

References

[1] Parsec. `https://hackage.haskell.org/package/parsec`.

[2] Runtypes. `https://github.com/pelotom/runtypes`.

[3] Wit. `https://wit.ai`.

[4] Microsoft. TypeScript. `https://www.typescriptlang.org`.

[5] Alex Renda, Harrison Goldstein, Sarah Bird, Chris Quirk, and Adrian Sampson. Abstractions for ai-based user interfaces and systems, 2017.