A Framework for Creating Natural Language User Interfaces for Action-Based Applications

Stephen Chong
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA
schong@cs.cornell.edu

Riccardo Pucella
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA
riccardo@cs.cornell.edu

Abstract

In this paper we present a framework for creating natural language interfaces to action-based applications. Our framework uses a number of reusable application-independent components, in order to reduce the effort of creating a natural language interface for a given application. Using a type-logical grammar, we first translate natural language sentences into expressions in an extended higher-order logic. These expressions can be seen as executable specifications corresponding to the original sentences. The executable specifications are then interpreted by invoking appropriate procedures provided by the application for which a natural language interface is being created.

1 INTRODUCTION

The separation of the user interface from the application is regarded as a sound design principle. A clean separation of these components allows different user interfaces such as GUI, command-line and voice-recognition interfaces. To support this feature, an application would supply an *application interface*. Roughly speaking, an application interface is a set of "hooks" that an application provides so that user interfaces can access the application's functionality. A user interface issues commands and queries to the application through the application interface; the application executes these commands and queries, and returns the results back to the user interface. We are interested in applications whose interface can be described in terms of actions that modify the application's state, and predicates that query the current state of the application. We refer to such applications as *action-based applications*.

In this paper, we propose a framework for creating natural language user interfaces to action-based applications. These user interfaces will accept commands from the user in the form of natural language sentences. We do not address how the user inputs these sentences (by typing, by speaking into a voice recognizer, etc), but rather focus on what to do with those sentences. Intuitively, we translate natural language sentences into appropriate calls to procedures available through the application interface.

As an example, consider the application TOYBLOCKS. It consists of a graphical representation of two blocks on a table, that can be moved, and put one on top of the other. We would like to be able to take a sentence such as *move block one on block two*, and have it translated into suitable calls to the TOYBLOCKS interface that would move block 1 on top of block 2. (This requires that the interface of TOYBLOCKS supplies a procedure for moving blocks.) While this example is simple, it already exposes most of the issues with which our framework must deal.

Our framework architecture is sketched in Figure 1. The diagram shows an application with several different user interfaces. The box labeled "NLUI" represents the natural language user interface that our framework is designed to implement. Our framework is appropriate for applications that provide a suitable application interface, which is described in Section 2. We expect that most existent applications will not provide an interface conforming to our requirements. Thus, an *adapter* might be required, as shown in the figure. Other user interfaces can also build on this application interface. The user interface labeled "Other UI 1" (for instance, a command-line interface) does just that. The application may have some user

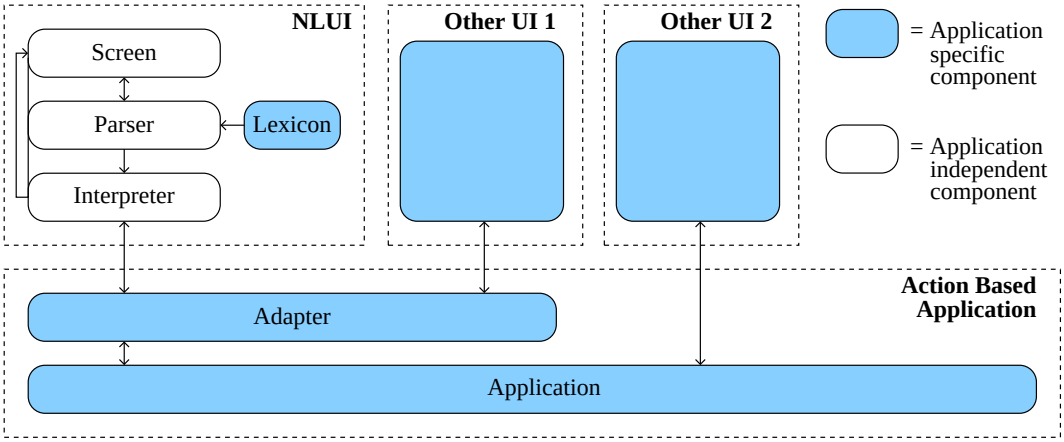

Figure 1: Architecture

interfaces that interact with the application through other means, such as the user interface "Other UI 2" (for instance, the native graphical interface of the application).

The translation from natural language sentences to application interface calls is achieved in two steps. The first step is to use a categorial grammar (Carpenter, 1997) to derive an intermediate representation of the semantics of the input sentence. An interesting feature of categorial grammars is that the semantics of the sentence is compositionally derived from the meaning of the words in the lexicon. The derived meaning is a formula of higher-order logic (Andrews, 1986). The key observation is that such a formula can be seen as an *executable specification*. More precisely, it corresponds to an expression of a simply-typed λ-calculus (Barendregt, 1981). The second step of our translation is to execute this λ-calculus expression via calls to procedures supplied by the application interface.

We implement the above scheme as follows. A parser accepts a natural language sentence from the user, and attempts to parse it using the categorial grammar rules and the vocabulary from the application-specific lexicon. The parser fails if it is not able to provide a unique unambiguous parsing of the sentence. Successful parsing results in a formula in our higher-order logic, which corresponds to an expression in an action calculus—a λ-calculus equipped with a notion of action. This expression is passed to the action calculus interpreter, which "executes" the expression by making appropriate calls to the application via the application interface. The interpreter may report back to the screen the results of executing the actions.

The main advantage of our approach is its modularity. This architecture contains only a few application-specific components, and has a number of reusable components. More precisely, the categorial grammar parser and the action calculus interpreter are generic and reusable across different applications. The lexicon, on the other hand, provides an application-specific vocabulary, and describes the semantics of the vocabulary in terms of a specific application interface.

In Section 2 we describe our requirements for action-based applications. We define the notion of an application interface, and provide a semantics for such an interface in terms of a model of the application. In Section 3 we present an action calculus that can be used to capture the meaning of imperative natural language sentences. The semantics of this action calculus are given in terms of an action-based application interface and application model; these semantics permit us to evaluate expressions of the action calculus by making calls to the application interface. Section 4 provides a brief introduction to categorial grammars. Section 5 shows how these components (action-based applications, action calculus, and categorial grammar) are used in our framework. We discuss some extensions to the framework in Section 6, and conclude in Section 7.

2 Action-Based Applications

Our framework applies to applications that provide a suitable Application Programmer Interface (API). Roughly speaking, such an interface provides procedures that are callable from external processes to

"drive" the application. In this section, we describe in detail the kind of interface needed by our approach. We also introduce a model of applications that will let us reason about the suitability of the whole framework.

2.1 APPLICATION INTERFACE

Our framework requires action-based applications to have an application interface that specifies which externally callable procedures exist in the application. This interface is meant to specify procedures that can be called from programs written in fairly arbitrary programming languages. To achieve this, we assume only that the calling language can distinguish between objects (the term 'object' is used is a nontechnical sense, to denote arbitrary data values), and Boolean values *#t* (true) and *#f* (false).

An application interface specifies the existence of a number of different kind of procedures.

(1) **Constants:** There is a set of constants representing objects of interest. For TOYBLOCKS, the constants are $\mathbf{b1}$, $\mathbf{b2}$, and $\mathbf{table}$.

(2) **Predicates:** There is a set of predicates defined over the states of the application. A predicate can be used to check whether objects satisfy certain properties, dependent on the state of the application. Predicates return truth values. For TOYBLOCKS, we consider the single predicate $\mathbf{is\_on}(bl, pos)$, that checks whether a particular block bl is in a particular position pos (on another block or on the table). Each predicate p has an associated arity, indicating how many arguments it needs.

(3) **Actions:** Finally, there is a set of actions defined by the application. Actions essentially effect a state change. Actions can be given arguments, for example, to effect a change to a particular object. For TOYBLOCKS, we consider a single action, $\mathbf{move}(bl, pos)$, which moves block bl to position pos (on another block or on the table). As with predicates, each action has an associated arity, which may be 0, indicating that the action is parameterless.

We emphasize that the application interface simply gives the names of the procedures that are callable by external processes. It does not actually define an implementation for these procedures.

In order to prevent predicates and actions from being given inappropriate arguments, we need some information about the actual kind of objects associated with constants, and that the predicates and actions take as arguments. We make the assumption that every object in the application belongs to at least one of many *classes* of objects. Let $\mathcal{C}$ be such a set of classes. Although this terminology evokes object-oriented programming, we emphasize that an object-oriented approach is not necessary for such interfaces; a number of languages and paradigms are suitable for implementing application interfaces.

We associate "class information" to every name in the interface via a map σ. More specifically, we associate with every constant c a set $\sigma(c) \subseteq \mathcal{C}$ representing the classes of objects that can be associated with c. We associate with each predicate p a set $\sigma(p) \subseteq \mathcal{C}^n$ (where n is the arity of the predicate), indicating for which classes of objects the predicate is defined. Similarly, we associate with each action a a set $\sigma(a) \subseteq \mathcal{C}^n$ (again, where n is the arity of the action, which in this case can be 0). As we will make clear shortly, we only require that the application return meaningful values for objects of the right classes.

Formally, an application interface is a tuple $I = (C, P, A, \mathcal{C}, \sigma)$, where C is a set of constant names, P is a set of predicate names, A is a set of action names, $\mathcal{C}$ is the set of classes of the application, and σ is the map associating every element of the interface with its corresponding class information. The procedures in the interface provide a means for an external process to access the functionality of the application, by presenting to the language a generally accessible version of the constants, predicates, and actions. Of course, in our case, we are not interested in having arbitrary processes invoking procedures in the interface, but specifically an interpreter that interprets commands written in a natural language.

In a precise sense, the map σ describes typing information for the elements of the interface. However, because we do not want to impose a particular type system on the application (for instance, we do not want to assume that the application is object-oriented), we instead assume a form of dynamic typing. More precisely, we assume that there is a way to check if an object belongs to a given class. This can either be performed through special *guard predicates* in the application interface (for instance, a procedure $\mathbf{is\_block}$ that returns true if the supplied object is actually a block), or a mechanism similar to Java's *instanceOf* operator.

Example 2.1: As an example, consider the following interface I_T for TOYBLOCKS. Let $I_T = (C, P, A, \mathcal{C}, \sigma)$, where, as we discussed earlier,

$$C = \{\mathbf{b1}, \mathbf{b2}, \mathbf{table}\}$$
$$P = \{\mathbf{is\_on}\}$$
$$A = \{\mathbf{move}\}$$

We consider only two classes of objects, *block*, representing the blocks that can be moved, and *position*, representing locations where blocks can be located. Therefore, $\mathcal{C} = \{block, position\}$.

To define σ, consider the way in which the interface could be used. The constant $\mathbf{b1}$ represents an object that is both a block that can be moved, and a position to which the other block can be moved to (since we can stack blocks on top of each other). The constant $\mathbf{b2}$ is similar. The constant $\mathbf{table}$ represents an object that is a position only. Therefore, we have:

$$\sigma(\mathbf{b1}) = \{block, position\}$$
$$\sigma(\mathbf{b2}) = \{block, position\}$$
$$\sigma(\mathbf{table}) = \{position\}.$$

Correspondingly, we can derive the class information for $\mathbf{is\_on}$ and $\mathbf{move}$:

$$\sigma(\mathbf{is\_on}) = \{(block, position)\}$$
$$\sigma(\mathbf{move}) = \{(block, position)\}. \blacksquare$$

2.2 APPLICATION MODEL

In order to reason formally about the interface, we provide a semantics to the procedures in the interface. This is done by supplying a model of the underlying application. We make a number of simplifying assumptions about the application model, and discuss relaxing some of these assumptions in Section 6.

Applications are modeled using four components:

(1) **Interface:** The interface, as we saw in the previous section, specifies the procedures that can be used to query and affect the application. The interface also defines the set $\mathcal{C}$ of classes of objects in the application.

(2) **States:** A state is, roughly speaking, everything that is relevant to understand how the application behaves. At any given point in time, the application is in some state. We assume that an application's state changes only through explicit actions.

(3) **Objects:** This defines the set of objects that can be manipulated, or queried, in the application. As we already mentioned, we use the term 'object' in the generic sense, without implying that the application is implemented through an object-oriented language. Every object is associated with at least one class.

(4) **Interpretation:** An interpretation associates with every element of the interface a "meaning" in the application model. As we shall see, it associates with every constant an object of the model, with every predicate a predicate on the model, and with every action a state-transformation on the model.

Formally, an application is a tuple $M = (I, \mathcal{S}, \mathcal{O}, \pi)$, where I in an interface (that defines the constants, predicates, and actions of the application, as well as the classes of the objects), $\mathcal{S}$ is the set of states of the application, $\mathcal{O}$ is the set of objects, and π is the interpretation.

We extend the map σ defined in the interface to also provide class information for the objects in $\mathcal{O}$. Specifically, we define for every object $o \in \mathcal{O}$ a set $\sigma(o) \subseteq \mathcal{C}$ of classes to which that object belongs. An object can belong to more than one class.

The map π associates with every state and every element in the interface (i.e., every constant, predicate and action) the appropriate interpretation of that element at that state. Specifically, for a state $s \in \mathcal{S}$, we have $\pi(s)(c) \in \mathcal{O}$. Therefore, constants can denote different objects at different states of the applications. For predicates, $\pi(s)(p)$ is a partial function from $\mathcal{O} \times \ldots \times \mathcal{O}$ to truth values *tt* or *ff*. This means that predicates are *pure*, in that they do not modify the state of an application; they are simply used to query

the state. For actions, $\pi(s)(a)$ is a partial function from $\mathcal{O} \times \ldots \times \mathcal{O}$ to $\mathcal{S}$. The interpretation π is subject to the following conditions. For a given predicate p, the interpretation $\pi(s)(p)$ must be defined on objects of the appropriate class. Thus, the domain of the partial function $\pi(s)(p)$ must at least consist of $\{(o_1, \ldots, o_n) \mid \sigma(o_1) \times \ldots \times \sigma(o_n) \cap \sigma(p) \neq \varnothing\}$. Similarly, for a given action a, the domain of the partial function $\pi(s)(a)$ must at least consist of $\{(o_1, \ldots, o_n) \mid \sigma(o_1) \times \ldots \times \sigma(o_n) \cap \sigma(a) \neq \varnothing\}$. Furthermore, any class associated with a constant must also be associated with the corresponding object. In other words, for all constants c, we must have $\sigma(c) \subseteq \sigma(\pi(s)(c))$ for all states s.

Example 2.2: We give a model M_T for our sample TOYBLOCKS application, to go with the interface I_T defined in Example 2.1. Let $M_T = (I_T, \mathcal{S}, \mathcal{O}, \pi)$. We will consider only three states in the application, $\mathcal{S} = \{s_1, s_2, s_3\}$, which can be described variously:

in state s_1, blocks 1 and 2 are on the table
in state s_2, block 1 is on block 2, and block 2 is on the table
in state s_3, block 1 is on the table, and block 2 is on block 1.

We consider only three objects in the model, $\mathcal{O} = \{b_1, b_2, t\}$, where b_1 is block 1, b_2 is block 2, and t is the table. We extend the map σ in the obvious way:

$$\sigma(b_1) = \{block, position\}$$
$$\sigma(b_2) = \{block, position\}$$
$$\sigma(t) = \{position\}$$

The interpretation for constants is particularly simple, as the interpretation is in fact independent of the state (in other words, the constants refer to the same objects at all states).

$$\pi(s)(\mathbf{b1}) = b_1$$
$$\pi(s)(\mathbf{b2}) = b_2$$
$$\pi(s)(\mathbf{table}) = t.$$

The interpretation of the **is_on** predicates is straightforward:

$$\pi(s_1)(\mathbf{is\_on})(x) = \begin{cases} \mathit{tt} & \text{if } x \in \{(b_1, t), (b_2, t)\} \\ \mathit{ff} & \text{if } x \in \{(b_1, b_1), (b_1, b_2), (b_2, b_1), (b_2, b_2)\} \end{cases}$$
$$\pi(s_2)(\mathbf{is\_on})(x) = \begin{cases} \mathit{tt} & \text{if } x \in \{(b_1, b_2), (b_2, t)\} \\ \mathit{ff} & \text{if } x \in \{(b_1, t), (b_1, b_1), (b_2, b_1), (b_2, b_2)\} \end{cases}$$
$$\pi(s_3)(\mathbf{is\_on})(x) = \begin{cases} \mathit{tt} & \text{if } x \in \{(b_1, t), (b_2, b_1)\} \\ \mathit{ff} & \text{if } x \in \{(b_1, b_1), (b_1, b_2), (b_2, t), (b_2, b_2)\}. \end{cases}$$

The interpretation of **move** is also straightforward:

$$\pi(s_1)(\mathbf{move})(x) = \begin{cases} s_2 & \text{if } x = (b_1, b_2) \\ s_3 & \text{if } x = (b_2, b_1) \\ s_1 & \text{if } x \in \{(b_1, t), (b_1, b_1), (b_2, t), (b_2, b_2)\} \end{cases}$$
$$\pi(s_2)(\mathbf{move})(x) = \begin{cases} s_1 & \text{if } x = (b_1, t) \\ s_2 & \text{if } x \in \{(b_1, b_1), (b_1, b_2), (b_2, t), (b_2, b_1), (b_2, b_2)\} \end{cases}$$
$$\pi(s_3)(\mathbf{move})(x) = \begin{cases} s_1 & \text{if } x = (b_2, t) \\ s_3 & \text{if } x \in \{(b_1, t), (b_1, b_1), (b_1, b_2), (b_2, b_1), (b_2, b_2)\} \end{cases}$$

If a block is unmovable (that is, if there is another block on it), then the state does not change following a move operation. ∎

3 AN ACTION CALCULUS

Action-based application interfaces are designed to provide a means for external processes to access the functionality of an application. In this section we define a powerful and flexible language that can be interpreted as calls to an application interface. The language we use is a simply-typed λ-calculus extended with a notion of action. It is effectively a computational λ-calculus in the style of Moggi (1989), although

we give a nonstandard presentation in order to simplify expressing the language semantics in terms of an application interface.

The calculus is parameterized by a particular application interface and application model. The application interface provides the primitive constants, predicates, and actions, that can be used to build more complicated expressions, while the application model is used to define the semantics.

3.1 SYNTAX

Every expression in the language is given a type, intuitively describing the kind of values that the expression produces. The types used in this language are given by the following grammar.

Types:

τ ::=	type
Obj	object
Bool	boolean
Act	action
$\tau_1 \rightarrow \tau_2$	function

The types τ correspond closely to the types required by the action-based application interfaces we defined in the previous section: the type Bool is the type of truth values, with constants **true** and **false** corresponding to the Boolean values *tt* and *ff*, and the type Obj is the type of generic objects. The type Act is more subtle; an expression of type Act represents an action that can be executed to change the state of the application. This is an example of computational type as defined by Moggi (1989). As we shall see shortly, expressions of type Act can be interpreted as calls to the action procedures of the application interface.

The classes $\mathcal{C}$ defined by the application interface have no corresponding types in this language—instead, all objects have the type Obj. Incorporating these classes as types is an obvious possible extension (see Section 6).

The syntax of the language is a straightforward extension of that of the λ-calculus.

Syntax of Expressions:

v ::=	value
true | **false**	boolean
$\lambda x{:}\tau.e$	function
skip	null action
e ::=	expression
x	variable
v	value
$id_c()$	constant
$id_p(e_1, \ldots, e_n)$	predicate
$id_a(e_1, \ldots, e_n)$	action
$e_1\ e_2$	application
$e_1?e_2 : e_3$	conditional
$e_1; e_2$	action sequencing

The expressions $id_c()$, $id_p(e_1, \ldots, e_n)$ and $id_a(e_1, \ldots, e_n)$ correspond to the procedures (respectively, constants, predicates, and actions) available in the application interface.[1] So, for TOYBLOCKS, the constants are **b1**, **b2**, and **table**; the only predicate is **is_on**; and the only action is **move**. The expression $e_1?e_2 : e_3$ is a conditional expression, evaluating to e_2 if e_1 evaluates to **true**, and e_3 if e_1 evaluates to **false**. The expression $e_1; e_2$ (when e_1 and e_2 are actions) evaluates to an action corresponding to performing e_1 followed by e_2. The constant **skip** represents an action that has no effect.

[1] Constants are written $id_c()$ as a visual reminder that they are essentially functions: $id_c()$ may yield different values at different states, as the semantics will make clear.

Example 3.1: Consider the interface for TOYBLOCKS. The expression $\mathbf{b1}()$ represents block 1, while $\mathbf{table}()$ represents the table. The expression $\mathbf{move}(\mathbf{b1}(), \mathbf{table}())$ represents the action of moving block 1 on the table. Similarly, the action $\mathbf{move}(\mathbf{b1}(), \mathbf{table}()); \mathbf{move}(\mathbf{b2}(), \mathbf{b1}())$ represents the composite action of moving block 1 on the table, and then moving block 2 on top of block 1. ∎

3.2 OPERATIONAL SEMANTICS

The operational semantics is defined with respect to the application model. More precisely, the semantics is given by a transition relation, written $(s, e) \longrightarrow (s', e')$, where s, s' are states of the application, and e, e' are expressions. Intuitively, this represents the expression e executing in state s, and making a one-step transition to a (possibly different) state s' and a new expression e'.

To accommodate the transition relation, we need to extend the syntax of expressions to account for object values produced during the evaluation. We also include a special value $\star$ that represents an exception raised by the code. This exception is used to capture various errors that may occur during evaluation.

Additional Syntax of Expressions:

$v_o \in \mathcal{O}$	object value
$v ::=$	value
...	
v_o	object
$\star$	exception

The transition relation is parameterized by the functions δ_c, δ_p and δ_p, given below. These functions provide a semantics to the constant, predicate, and action procedures respectively, and are derived from the interpretation π in the application model. The intuition is that evaluating these functions corresponds to making calls to the appropriate procedures on the given application interface, and returning the result.

Reduction Rules for Interface Elements:

$$\delta_c(s, id_c) \triangleq \pi(s)(id_c)$$

$$\delta_p(s, id_p, v_1, \ldots, v_n) \triangleq \begin{cases} \pi(s)(id_p)(v_1, \ldots, v_n) & \text{if } \sigma(v_1) \times \ldots \times \sigma(v_n) \cap \sigma(id_p) \neq \varnothing \\ \star & \text{otherwise} \end{cases}$$

$$\delta_a(s, id_a, v_1, \ldots, v_n) \triangleq \begin{cases} \pi(s)(id_a)(v_1, \ldots, v_n) & \text{if } \sigma(v_1) \times \ldots \times \sigma(v_n) \cap \sigma(id_a) \neq \varnothing \\ \star & \text{otherwise} \end{cases}$$

Note that determining whether or not a primitive throws an exception depends on being able to establish the class of an object (via the map σ). We can thus ensure that we never call an action or predicate procedure on the application interface with inappropriate objects, and so we guarantee a kind of dynamic type-safety with respect to the application interface.

Reduction Rules:

(Red App 1)
$$\frac{(s, e_1) \longrightarrow (s, e_1')}{(s, e_1\ e_2) \longrightarrow (s, e_1'\ e_2)}$$

(Red App 2)
$$\frac{(s, e_1) \longrightarrow (s, \star)}{(s, e_1\ e_2) \longrightarrow (s, \star)}$$

(Red App 3)
$$\frac{}{(s, (\lambda x{:}\tau.e_1)\ e_2) \longrightarrow (s, e_1\{x \leftarrow e_2\})}$$

(Red OCon)
$$\frac{}{(s, id_c()) \longrightarrow (s, \delta_c(s, id_c))}$$

(Red PCon 1)
$$\frac{(s, e_i) \longrightarrow (s, e_i') \quad \text{for some } i \in [1..n]}{(s, id_p(\ldots, e_i, \ldots)) \longrightarrow (s, id_p(\ldots, e_i', \ldots))}$$

(Red PCon 2)
$$\frac{(s, e_i) \longrightarrow (s, \star) \quad \text{for some } i \in [1..n]}{(s, id_p(e_1, \ldots, e_n)) \longrightarrow (s, \star)}$$

(Red PCon 3)
$$\frac{}{(s, id_p(v_1, \ldots, v_n)) \longrightarrow (s, v)} \quad \delta_p(s, id_p, v_1, \ldots, v_n) = v$$

(Red If 1) $\dfrac{(s, e_1) \longrightarrow (s, e_1')}{(s, e_1 ? e_2 : e_3) \longrightarrow (s, e_1' ? e_2 : e_3)}$ (Red If 2) $\dfrac{(s, e_1) \longrightarrow (s, \star)}{(s, e_1 ? e_2 : e_3) \longrightarrow (s, \star)}$ (Red If 3) $\dfrac{}{(s, v ? e_{\mathbf{true}} : e_{\mathbf{false}}) \longrightarrow (s, e_v)}$

(Red Seq 1) $\dfrac{(s, e_1) \longrightarrow (s', e_1')}{(s, e_1; e_2) \longrightarrow (s', e_1'; e_2)}$ (Red Seq 2) $\dfrac{}{(s, \star; e) \longrightarrow (s, \star)}$ (Red Seq 3) $\dfrac{}{(s, \mathbf{skip}; e) \longrightarrow (s, e)}$

(Red ACon 1) $\dfrac{(s, e_i) \longrightarrow (s, e_i') \quad \text{for some } i \in [1..n]}{(s, id_a(\ldots, e_i, \ldots)) \longrightarrow (s, id_a(\ldots, e_i', \ldots))}$ (Red ACon 2) $\dfrac{(s, e_i) \longrightarrow (s, \star) \quad \text{for some } i \in [1..n]}{(s, id_a(e_1, \ldots, e_n)) \longrightarrow (s, \star)}$

(Red ACon 3)

$$\frac{}{(s, id_a(v_1, \ldots, v_n)) \longrightarrow (s', \mathbf{skip})} \quad \delta_a(s, id_a, v_1, \ldots, v_n) = s'$$

(Red ACon 4)

$$\frac{}{(s, id_a(v_1, \ldots, v_n)) \longrightarrow (s, \star)} \quad \delta_a(s, id_a, v_1, \ldots, v_n) = \star$$

The operational semantics is a combination of call-by-name and call-by-value semantics. More specifically, actions are evaluated in a call-by-name fashion, while the remainder of the language is evaluated in a call-by-value fashion. Intuitively, actions are evaluated under call-by-name because premature evaluation of actions could lead to action procedures in the application interface being called inappropriately. For example, under call-by-value semantics for actions, evaluation of the following expression would call the action procedure for $\mathbf{A}$, assuming $\mathbf{A}$ is an action in the application interface: $(\lambda x{:}\mathsf{Act}.\mathbf{false}?x{:}\mathbf{skip})\ \mathbf{A}$. This does not agree with the intuitive interpretation of actions. More importantly, the mapping from natural language sentences to expressions in our calculus naturally yields a call-by-name interpretation of actions.

3.3 TYPE SYSTEM

We use type judgments to ensure that expressions are assigned types appropriately, and that the types themselves are *well-formed*. Roughly speaking, a type is well-formed if it preserves the separation between pure computations (computations with no side-effects) and imperative computations (computations that may have side-effects). The type system enforces that pure computations do not change the state of the application. This captures the intuition that declarative sentences—corresponding to pure computations—should not change the state of the world. (This correspondence between declarative sentences and pure computations is made clear in the next section.) The rules for the type well-formedness judgment $\vdash \tau$ ok are given in the following table.

Judgments $\vdash \tau$ pure **,** $\vdash \tau$ imp **, and** $\vdash \tau$ ok**:**

(Pure Obj) $\dfrac{}{\vdash \mathsf{Obj}\ \text{pure}}$ (Pure Bool) $\dfrac{}{\vdash \mathsf{Bool}\ \text{pure}}$ (Pure Fun) $\dfrac{\vdash \tau_1 \text{ pure} \quad \vdash \tau_2 \text{ pure}}{\vdash \tau_1 \to \tau_2 \text{ pure}}$

(Imp Act) $\dfrac{}{\vdash \mathsf{Act}\ \text{imp}}$ (Imp Fun) $\dfrac{\vdash \tau_2 \text{ imp}}{\vdash \tau_1 \to \tau_2 \text{ imp}}$ (OK Pure) $\dfrac{\vdash \tau \text{ pure}}{\vdash \tau \text{ ok}}$ (OK Imperative) $\dfrac{\vdash \tau \text{ imp}}{\vdash \tau \text{ ok}}$

The judgment $\Gamma \vdash e : \tau$ assigns a type τ to expression e in a well-formed environment Γ. An environment Γ defines the types of all variables in scope. An environment is of the form $x_1 : \tau_1, \ldots, x : n : \tau_n$, and defines each variable x_i to have type τ_i. We require that variables do not repeat in a well-formed environment. The typing rules for expressions are essentially standard, with the exception of the typing rule for functions, which requires that function types $\tau \to \tau'$ be well-formed.

Judgment $\Gamma \vdash e : \tau$:

(Typ Var) $\dfrac{}{\Gamma, x : \tau \vdash x : \tau}$ (Typ Obj) $\dfrac{}{\Gamma \vdash v_o : \mathsf{Obj}}$ (Typ True) $\dfrac{}{\Gamma \vdash \mathbf{true} : \mathsf{Bool}}$ (Typ False) $\dfrac{}{\Gamma \vdash \mathbf{false} : \mathsf{Bool}}$ (Typ Exc) $\dfrac{}{\Gamma \vdash \star : \tau}$

(Typ App) $\dfrac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$ (Typ Fun) $\dfrac{\Gamma, x : \tau \vdash e : \tau' \quad \vdash \tau \to \tau' \text{ ok}}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'}$

(Typ If) $\dfrac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau}$ (Typ Skip) $\dfrac{}{\Gamma \vdash \mathbf{skip} : \mathsf{Act}}$ (Typ Seq) $\dfrac{\Gamma \vdash e_1 : \mathsf{Act} \quad \Gamma \vdash e_2 : \mathsf{Act}}{\Gamma \vdash e_1; e_2 : \mathsf{Act}}$

(Typ ACon) $\dfrac{\Gamma \vdash e_i : \mathsf{Obj} \quad \forall i \in [1..n]}{\Gamma \vdash id_a(e_1, \ldots, e_n) : \mathsf{Act}}$ (Typ OCon) $\dfrac{}{\Gamma \vdash id_c() : \mathsf{Obj}}$ (Typ PCon) $\dfrac{\Gamma \vdash e_i : \mathsf{Obj} \quad \forall i \in [1..n]}{\Gamma \vdash id_p(e_1, \ldots, e_n) : \mathsf{Bool}}$

It is straightforward to show that our type system is sound, that is, that type-correct expressions do not get stuck when evaluating.

Theorem 3.2: *If $\vdash e : \tau$, and s is a state, then there exists a state s' and value v such that $\vdash v : \tau$ and $(s, e) \longrightarrow^* (s', v)$. Moreover, if $\vdash \tau$ pure, then $s' = s$.*

Theorem 3.2 also implies that the language is strongly normalizing: the evaluation of every expression terminates. This is a very desirable property for the language, since it will form part of the user interface.

Example 3.3: Consider the following example, interpreted with respect to the application model of Example 2.2. In state s_1 (where both block 1 and 2 are on the table), let us trace through the execution of the expression $(\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x, y))\ (\mathbf{b1}())\ (\mathbf{b2}())$. (We omit the derivation indicating how each step is justified.)

$$
\begin{array}{l}
(s_1, (\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x, y))\ (\mathbf{b1}())\ (\mathbf{b2}())) \longrightarrow \\
\quad (s_1, (\lambda y{:}\mathsf{Obj}.\mathbf{move}(\mathbf{b1}(), y))\ (\mathbf{b2}())) \longrightarrow \\
\quad (s_1, \mathbf{move}(\mathbf{b1}(), \mathbf{b2}())) \longrightarrow \\
\quad (s_1, \mathbf{move}(b_1, \mathbf{b2}())) \longrightarrow \\
\quad (s_1, \mathbf{move}(b_1, b_2)) \longrightarrow \\
\quad (s_2, \mathbf{skip}).
\end{array}
$$

In other words, evaluating the expression in state s_1 leads to state s_2, where indeed block 1 is on top of block 2. ∎

3.4 A Direct Interpreter

The main reason for introducing the action calculus of this section is to provide a language in which to write expressions invoking procedures available in the application interface. However, the operational semantics given above rely on explicitly passing around the state of the application. This state is taken from the application model. In the model, the state is an explicit datum that enters the interpretation of constants, predicates and actions. Of course, in the actual application, the state is implicitly maintained by the application itself. Invoking an action procedure on the application interface modifies the current state of the application, putting the application in a new state. This new state is not directly visible to the user.

We can implement an interpreter based on the above operational semantics but without carrying around the state explicitly. To see this, observe that the state is only relevant for the evaluation of the primitives (constants, predicates, and actions). More importantly, it is always the current state of the application that is relevant, and only actions are allowed to change the state. We can therefore implement an interpreter by simply directly invoking the procedures in the application interface when the semantics tells us to reduce

via δ_c, δ_p, or δ_a. Furthermore, we need to be able to raise an exception $\star$ if the objects passed to the interface are not of the right class. This requires querying for the class of an object. As we indicated in Section 2.1, we simply assume that this can be done, either through language facilities (an *instanceOf* operator), or through explicit procedures in the interface that check whether an object is of a given class.

In summary, given an application with a suitable application interface, we can write an interpreter for our action calculus that will interpret expressions by invoking procedures available through the application interface when appropriate. The interpreter does not require an application model. The model is useful to establish properties of the interpreter, and if one wants to reason about the execution of expressions via the above operational semantics.

4 CATEGORIAL GRAMMARS

In the last section, we introduced an action calculus that lets us write expressions that can be understood via calls to the application interface. The aim of this section is to use this action calculus as the target of a translation from natural language sentences. In other words, we describe a way to take a natural language sentence and produce a corresponding expression in our action calculus that captures the meaning of the sentence. Our main tool is categorial grammars.

Categorial grammars provide a mechanism to assign semantics to sentences in natural language in a compositional manner. As we shall see, we can obtain a compositional translation from natural language sentences into the action calculus presented in the previous section, and thus provide a simple natural language user interface for a given application. This section provides a brief exposition of categorial grammars, based on Carpenter's (1997) presentation. We should note that the use of categorial grammars is not a requirement in our framework. Indeed, any approach to provide semantics to natural language sentences in higher-order logic, which can be viewed as a simply-typed λ-calculus (Andrews, 1986), can be adapted to our use. For instance, Moortgat's (1997) multimodal categorial grammars, which can handle a wider range of syntactic constructs, can also be used for our purposes. To simplify the exposition, we use the simpler categorial grammars in this paper.

Categorial grammars were originally developed by Ajdukiewicz (1935) and Bar-Hillel (1953), and later generalized by Lambek (1958). The idea behind categorial grammars is simple. We start with a set of *categories*, each category representing a grammatical function. For instance, we can start with the simple categories *np* representing noun phrases, *pp* representing prepositional phrases, *s* representing declarative sentences and *a* representing imperative sentences. Given categories A and B, we can form the *functor* categories A/B and $B\backslash A$. The category A/B represents the category of syntactic units that take a syntactic unit of category B to their right to form a syntactic unit of category A. Similarly, the category $B\backslash A$ represents the category of syntactic units that take a syntactic unit of category B to their left to form a syntactic unit of category A.

Consider some examples. The category $np\backslash s$ is the category of intransitive verbs (e.g., *laughs*): they take a noun phrase on their left to form a sentence (e.g., *Alice laughs* or *the reviewer laughs*). Similarly, the category $(np\backslash s)/np$ represents the category of transitive verbs (e.g., *takes*): they take a noun phrase on their right and then a noun phrase on their left to form a sentence (e.g., *Alice takes the doughnut*).

The main goal of categorial grammars is to provide a method of determining the well-formedness of natural language. A *lexicon* associates every word (or complex sequence of words that constitute a single lexical entry) with one or more categories. The approach described by Lambek (1958) is to prescribe a calculus of categories so that if a sequence of words can be assigned a category A according to the rules, then the sequence of words is deemed a well-formed syntactic unit of category A. Hence, a sequence of words is a well-formed noun phrase if it can be shown in the calculus that it has category np. As an example of reduction, we see that if σ_1 has category A and σ_2 has category $A\backslash B$, then $\sigma_1\ \sigma_2$ has category B. Schematically, $A, A\backslash B \Rightarrow B$. Moreover, this goes both ways, that is, if $\sigma_1\ \sigma_2$ has category B and σ_1 can be shown to have category A, then we can derive that σ_2 has category $A\backslash B$.

Van Benthem (1986) showed that this calculus could be used to assign a semantics to terms by following the derivation of the categories. Assume that every basic category is assigned a type in our action calculus, through a type assignment T. A type assignment T can be extended to functor categories by putting $T(A/B) = T(B\backslash A) = T(B) \rightarrow T(A)$. The lexicon is extended so that every word is now associated with one or more pairs of a category A and an expression α in our action calculus of the appropriate type,

that is, $\vdash \alpha : T(A)$.

We use the sequent notation $\alpha_1 : A_1, \ldots, \alpha_n : A_n \Rightarrow \alpha : A$ to mean that expressions $\alpha_1, \ldots, \alpha_n$ of categories $A_1, \ldots, A_n$ can be concatenated to form an expression α of category A. We call $\alpha : A$ the conclusion of the sequent. We use capital Greek letters (Γ, Δ,...) to represent sequences of expressions and categories. We now give rules that allow us to derive new sequents from other sequents.

Categorial Grammar Sequent Rules:

(Seq Id)
$$\frac{}{\alpha : A \Rightarrow \alpha : A}$$

(Seq Cut)
$$\frac{\Gamma_2 \Rightarrow \beta : B \quad \Gamma_1, \beta : B, \Gamma_3 \Rightarrow \alpha : A}{\Gamma_1, \Gamma_2, \Gamma_3 \Rightarrow \alpha : A}$$

(Seq App Right)
$$\frac{\Delta \Rightarrow \beta : B \quad \Gamma_1, \alpha(\beta) : A, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \alpha : A/B, \Delta, \Gamma_2 \Rightarrow \gamma : C}$$

(Seq App Left)
$$\frac{\Delta \Rightarrow \beta : B \quad \Gamma_1, \alpha(\beta) : A, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \Delta, \alpha : B\backslash A, \Gamma_2 \Rightarrow \gamma : C}$$

(Seq Abs Right)
$$\frac{\Gamma, x : A \Rightarrow \alpha : B}{\Gamma \Rightarrow \lambda x.\alpha : B/A}$$

(Seq Abs Left)
$$\frac{x : A, \Gamma \Rightarrow \alpha : B}{\Gamma \Rightarrow \lambda x.\alpha : A\backslash B}$$

Example 4.1: Consider the following simple lexicon, suitable for the TOYBLOCKS application. The following types are associated with the basic grammatical units:

$T(np) = \mathsf{Obj}$
$T(pp) = \mathsf{Obj}$
$T(s) = \mathsf{Bool}$
$T(a) = \mathsf{Act}$

Here is a lexicon that captures a simple input language for TOYBLOCKS:

block one $\Rightarrow \mathbf{b1}() : \mathsf{np}$
block two $\Rightarrow \mathbf{b2}() : \mathsf{np}$
the table $\Rightarrow \mathbf{table}() : \mathsf{np}$
on $\Rightarrow (\lambda x{:}\mathsf{Obj}.x) : \mathsf{pp/np}$
is $\Rightarrow (\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{is\_on}(y, x)) : (\mathsf{np}\backslash\mathsf{s})/\mathsf{pp}$
if $\Rightarrow (\lambda x{:}\mathsf{Bool}.\lambda y{:}\mathsf{Act}.x?y : \mathbf{skip}) : (\mathsf{a}/\mathsf{a})/\mathsf{s}$
move $\Rightarrow (\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x, y)) : (\mathsf{a}/\mathsf{pp})/\mathsf{np}$

This is a particularly simple lexicon, since every entry is assigned a single term and category.

Using the above lexicon, the sentence *move block one on block two* can be associated with the string of expressions and categories $\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x, y) : (\mathsf{a}/\mathsf{pp})/\mathsf{np}$, $\mathbf{b1}() : \mathsf{np}$, $\lambda x{:}\mathsf{Obj}.x : \mathsf{pp}/\mathsf{np}$, $\mathbf{b2}() : \mathsf{np}$. The following derivation shows that this concatenation yields an expression of category a. (For reasons of spaces, we have elided the type annotations in λ-abstractions.)

$$\dfrac{\dfrac{}{\mathbf{b2}():\mathsf{np} \Rightarrow \mathbf{b2}():\mathsf{np}} \quad \dfrac{\dfrac{}{\mathbf{b1}():\mathsf{np} \Rightarrow \mathbf{b1}():\mathsf{np}} \quad (\dagger)}{\lambda x.\lambda y.\mathbf{move}(x,y):(\mathsf{a}/\mathsf{pp})/\mathsf{np}, \mathbf{b1}():\mathsf{np}, (\lambda x.x)\ (\mathbf{b2}()):\mathsf{pp} \Rightarrow (\lambda x.\lambda y.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x.x)\ (\mathbf{b2}())):\mathsf{a}}}{\lambda x.\lambda y.\mathbf{move}(x,y):(\mathsf{a}/\mathsf{pp})/\mathsf{np}, \mathbf{b1}():\mathsf{np}, \lambda x.x:\mathsf{pp}/\mathsf{np}, \mathbf{b2}():\mathsf{np} \Rightarrow (\lambda x.\lambda y.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x.x)\ (\mathbf{b2}())):\mathsf{a}}$$

where the subderivation $(\dagger)$ is simply:

$$(\dagger) : \dfrac{\dfrac{}{(\lambda x.x)\ (\mathbf{b2}()):\mathsf{pp} \Rightarrow (\lambda x.x)\ (\mathbf{b2}()):\mathsf{pp}} \quad \dfrac{}{(\lambda x.\lambda y.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x.x)\ (\mathbf{b2}())):\mathsf{a} \Rightarrow (\lambda x.\lambda y.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x.x)\ (\mathbf{b2}())):\mathsf{a}}}{(\lambda x.\lambda y.\mathbf{move}(x,y))\ (\mathbf{b1}()):\mathsf{a}/\mathsf{pp}, (\lambda x.x)\ (\mathbf{b2}()):\mathsf{pp} \Rightarrow (\lambda x.\lambda y.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x.x)\ (\mathbf{b2}())):\mathsf{a}}$$

Hence, the sentence is a well-formed imperative sentence. Moreover, the derivation shows that the meaning of the sentence *move block one on block two* is $(\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x{:}\mathsf{Obj}.x)\ (\mathbf{b2}()))$. The execution of this expression, similar to the one in Example 3.3, shows that the intuitive meaning of the sentence is reflected by the execution of the corresponding expression. ∎

One might hope that the expressions derived through a categorial grammar derivation are always valid expressions of our action calculus. To ensure that this property holds, we must somewhat restrict the kind of categories that can appear in a derivation. Call a category A *imperative* if it is category a, or if it is of the form B/C or $C\backslash B$ with B imperative. Let us say that a derivation *respects imperative structure* if every category A with an embedded category a that appears in the derivation is an imperative category. Intuitively, a derivation respects imperative structure if it cannot construct declarative sentences that depend on imperative subsentences, i.e., a declarative sentence cannot have any "side effects." We can show that any such derivation will produce expressions that typecheck in the type system of the previous section.

Theorem 4.2: *For any derivation of a sequent with conclusion $\alpha : A$ that respects imperative structure, we have $\vdash \alpha : T(A)$.*

So, given a natural language imperative sentence from the user, we use the lexicon to find the corresponding expressions and category pairs $\alpha_1 : A_1, \ldots, \alpha_n : A_n$, and then attempt to parse it, that is, to find a derivation for the sequent $\alpha_1 : A_1, \ldots, \alpha_n : A_n \Rightarrow \alpha : \mathsf{a}$. If a unique such derivation exists, then we have an unambiguous parsing of the natural language imperative sentence, and moreover, the action calculus expression α is the semantics of the imperative sentence.

5 PUTTING IT ALL TOGETHER...

We now have the major components of our framework: a model for action-based applications and interfaces to them; an action calculus which can be interpreted as calls to an application interface; and the use of categorial grammars to create expressions in our action calculus from natural language sentences.

Let's see how our framework combines these components by considering an end-to-end example for TOYBLOCKS. Suppose the user inputs the sentence *move block one on block two* when blocks 1 and 2 are both on the table. Our framework would process this sentence in the following steps.

(1) **Parsing:** The TOYBLOCKS lexicon is used to parse the sentence. Parsing succeeds only if there is a unique parsing of the sentence, otherwise the parsing step fails, because the sentence was either ambiguous, contained unknown words or phrases, or was ungrammatical. In this example, there is only a single parsing of the sentence (as shown in Example 4.1), and the result is the following expression in our action calculus, which has type Act:

$$(\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x,y))\ (\mathbf{b1}())\ \ ((\lambda x{:}\mathsf{Obj}.x)\ (\mathbf{b2}())).$$

(2) **Evaluating:** The action calculus expression is evaluated using a direct interpreter implementing the operational semantics of Section 3. The evaluation of the expression proceeds as follows.

$$\begin{array}{l}(s_1, (\lambda x{:}\mathsf{Obj}.\lambda y{:}\mathsf{Obj}.\mathbf{move}(x,y))\ (\mathbf{b1}())\ ((\lambda x{:}\mathsf{Obj}.x)\ (\mathbf{b2}()))) \longrightarrow \\ \quad (s_1, (\lambda y{:}\mathsf{Obj}.\mathbf{move}(\mathbf{b1}(),y))\ ((\lambda x{:}\mathsf{Obj}.x)\ (\mathbf{b2}()))) \longrightarrow \\ \quad (s_1, \mathbf{move}(\mathbf{b1}(), (\lambda x{:}\mathsf{Obj}.x)\ (\mathbf{b2}()))) \longrightarrow \\ \quad (s_1, \mathbf{move}(b_1, (\lambda x{:}\mathsf{Obj}.x)\ (\mathbf{b2}()))) \longrightarrow \\ \quad (s_1, \mathbf{move}(b_1, (\mathbf{b2}()))) \longrightarrow \\ \quad (s_1, \mathbf{move}(b_1, b_2)) \longrightarrow \\ \quad (s_2, \mathbf{skip}).\end{array}$$

In the process of this evaluation, several calls are generated to the application interface. In particular, calls are made to determine the identity of the object constants $\mathbf{b1}$ and $\mathbf{b2}$ as b_1 and b_2 respectively. Then, during the last transition, guard predicates such as $\mathbf{is\_block}(b_1)$ and $\mathbf{is\_position}(b_2)$ may be called to ensure that b_1 and b_2 are of the appropriate classes for being passed as arguments to $\mathbf{move}$. Since the objects are of the appropriate classes, the action $\mathbf{move}(b_1, b_2)$ is invoked via the application interface, and succeeds.

(3) **Reporting:** Following the evaluation of the expression, some result must be reported back to the user. Our framework does not detail what information is conveyed back to the user, but they must be informed if an exception was raised during the evaluation of the expression.

In this example, no exception was raised, so what to report to the user is at the discretion of the user interface. If the user interface had a graphical depiction of the state of TOYBLOCKS, it may now send queries to the application interface to determine the new state of the world, and modify its graphical display appropriately.

Let's consider what would happen if an exception ($\star$) was raised during the evaluation phase. For example, consider processing the sentence *move the table on block one*. The parsing phase would succeed, as the sentence is grammatically correct. However, prior to calling the action $\mathbf{move}(t, b_1)$, the evaluation would determine that the object t does not belong to the class $block$ (by a guard predicate such as $\mathbf{is\_block}(t)$ returning ff, or by some other mechanism). An exception would thus be raised, and some information must be reported back to the user during the reporting phase. Note that the framework has ensured that the action $\mathbf{move}(t, b_1)$ was not invoked on the application interface.

6 EXTENSIONS

Several extensions to this framework are possible. There is a mismatch of types in our framework. The application model permits a rich notion of types: any object of the application may belong to one or more classes. By contrast, our action calculus has a very simple notion of types, assigning the type Obj to all objects, and not statically distinguishing different classes of objects. The simplicity of our action calculus is achieved at the cost of dynamic type checking, which ensures that actions and predicates on the application interface are invoked only with appropriate parameters. It would be straightforward to extend the action calculus with a more refined type system that includes a notion of subtyping, to model the application classes. Not only would this extension remove many, if not all, of the dynamic type checks, but it may also reduce the number of possible parses of natural language sentences. The refined type system allows the semantics of the lexicon entries to be finer-grained, and by considering these semantics, some nonsensical parses of a sentence could be ignored. For example, in the sentence *pick up the book and the doughnut and eat it* the referent of *it* could naively be either *the book* or *the doughnut*; if the semantics of *eat* require an object of the class Food and the classes of *the book* and *the doughnut* are considered, then the former possibility could be ruled out.

Another straightforward extension to the framework is to allow the user to query the state by entering declarative sentences and treating them as yes-no interrogative sentences. For example, *block one is on the table?* This corresponds to accepting sequents of the form $\alpha_1 : A_1, \ldots, \alpha_n : A_n \Rightarrow \alpha : \mathsf{s}$, and executing the action calculus expression α, which has type Bool. The categorial grammar could be extended to accept other yes-no questions, such as *is block two on block one?* A more interesting extension (which would require a correspondingly more complex application model) is to allow hypothetical queries, such as *if you move block one on block two, is block one on the table?* This corresponds to querying *is block one on the table?* in the state that would result if the action *move block one on block two* were performed. This extension would bring our higher-order logic (that is, our action calculus) closer to dynamic logic (Groenendijk and Stokhof, 1991; Harel et al., 2000). It is not clear, however, how to derive a direct interpreter for such an extended calculus.

In Section 2.2 we made some simplifying assumptions about the application model. Chief among these assumptions was that an application's state changes only as a result of explicit actions. This assumption may be unrealistic if, for example, the application has multiple concurrent users. We can however extend the framework to relax this assumption. One way of relaxing it is to incorporate transactions into the application model and application interface: the application model would guarantee that within transactions, states change only as a result of explicit actions, but if no transaction is in progress then states may change arbitrarily. The evaluation of an action calculus expression would then be wrapped in a transaction.

Another restriction we imposed was that predicates be pure. It is of course technically possible to permit arbitrary state changes during the evaluation of predicates. In fact, we can modify the operational semantics to allow the evaluation of any expression to change states. If done properly, the key property is still preserved: the evaluation of constants, predicates or actions rely only on the current state, and all

other transitions do not rely on the state at all. Thus, the semantics remains consistent with interpreting expressions using calls to the application interface. However, doing this would lose the intuitive meaning of natural language sentences that do not contain actions; they should not change the state of the world.

7 Conclusion

We have presented a framework that simplifies the creation of simple natural language user interfaces for action-based applications. The key point of this framework is the use of a λ-calculus to mediate access to the application. The λ-calculus we define is used as a semantics for natural language sentences (via categorial grammars), and expressions in this calculus are executed by issuing calls to the application interface. The framework has a number of application-independent components, reducing the amount of effort required to create a simple natural language user interface for a given application.

A number of applications have natural language interfaces (Winograd, 1971; Price et al., 2000), but they appear to be designed specifically for the given application, rather than being a generic approach. A number of methodologies and frameworks exist for natural language interfaces for database queries (see Androutsopoulos et al. (1995) for a survey), but we are not aware of a framework for deriving natural language interfaces to general applications in a principled manner.

While the framework presented here is useful for the rapid development of simple natural language user interfaces, the emphasis is on simple. Categorial grammars (and other techniques that use higher order logic as the semantics of natural language) are limited in their ability to deal with the wide and diverse phenomena that occur in English. For example, additional mechanisms outside of the categorial grammar, probably application-specific, would be required to deal with discourse. However, categorial grammars are easily extensible, by expanding the lexicon, and many parts of the lexicon of a categorial grammar are reusable in different applications, making it well-suited to a framework for rapid development of natural language user interfaces.

It may seem that a limitation of our framework is that it is only suitable for applications for which we can provide an interface of the kind described in Section 2—the action calculus of Section 3 is specifically designed to be interpreted as calls to an action-based application. However, all the examples we considered can be provided with such an interface. It is especially interesting to note that our definition of action-based application interfaces is compatible with the notion of interface for XML web services (Barclay et al., 2002). This suggests that it may be possible to derive a natural language interface to XML Web Services using essentially the approach we advocate in this paper.

Acknowledgments

Thanks to Eric Breck and Vicky Weissman for comments on earlier drafts of this paper. This work was partially supported by NSF under grant CTC-0208535, by ONR under grants N00014-00-1-03-41 and N00014-01-10-511, and by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the ONR under grant N00014-01-1-0795.

References

Ajdukiewicz, K. (1935). Die syntaktische Konnexität. *Studia Philosophica*, 1:1–27.

Andrews, P. B. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press.

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

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58.

Barclay, T., Gray, J., Strand, E., Ekblad, S., and Richter, J. (2002). TerraService.NET: An introduction to web services. Technical Report MS–TR–2002–53, Microsoft Research.

Barendregt, H. P. (1981). *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, Amsterdam.

Carpenter, B. (1997). *Type-Logical Semantics*. The MIT Press.

Groenendijk, J. and Stokhof, M. (1991). Dynamic predicate logic. *Linguistics and Philosophy*, 14(1):39–100.

Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic Logic*. The MIT Press, Cambridge, Massachusetts.

Lambek, J. (1958). The mathematics of sentence structure. *The American Mathematical Monthly*, 65:154–170.

Moggi, E. (1989). Computational lambda-calculus and monads. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press.

Moortgat, M. (1997). Categorial type logics. In van Benthem, J. and ter Meulen, A., editors, *Handbook of Logic and Language*, chapter 2, pages 93–177. The MIT Press / Elsevier.

Price, D., Rilofff, E., Zachary, J. L., and Harvey, B. (2000). NaturalJava: a natural language interface for programming in Java. In *Intelligent User Interfaces*, pages 207–211.

van Benthem, J. (1986). The semantics of variety in categorial grammar. In Buszkowski, W., van Benthem, J., and Marciszewski, W., editors, *Categorial Grammar*, number 25 in Linguistics and Literary Studies in Eastern Europe, pages 37–55. John Benjamins. Previously appeared as Report 83-29, Department of Mathematics, Simon Fraser University (1983).

Winograd, T. (1971). Procedures as a representation for data in a computer program for understanding natural languages. Project MAC technical report MAC-TR-84, MIT.