Compositional desires as compositional programs

by

Joey Velez-Ginorio

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Brain and Cognitive Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .
Department of Brain and Cognitive Sciences
April 16, 2021

Certified by. . . . .
Nada Amin
Assistant Professor of Computer Science, Harvard University
Thesis Supervisor

Accepted by . . . .
Rebecca Saxe
John W. Jarve (1978) Professor of Brain and Cognitive Sciences, MIT

Compositional desires as compositional programs

by

Joey Velez-Ginorio

Submitted to the Department of Brain and Cognitive Sciences
on April 16, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Brain and Cognitive Sciences

Abstract

We routinely act in the world. And generally, we're not random actors. But rather actors directed at what we want, our desires. As observers we also assume this about others, and use the way others act in order to infer what they want. This work defines, explores, and tests a probabilistic programming language called Act. Our concern is whether humans act on and infer desires in the way Act does. In other words, do humans act like Act?

Thesis Supervisor: Nada Amin
Title: Assistant Professor of Computer Science, Harvard University

Contents

List of Figures

Chapter 1

Introduction

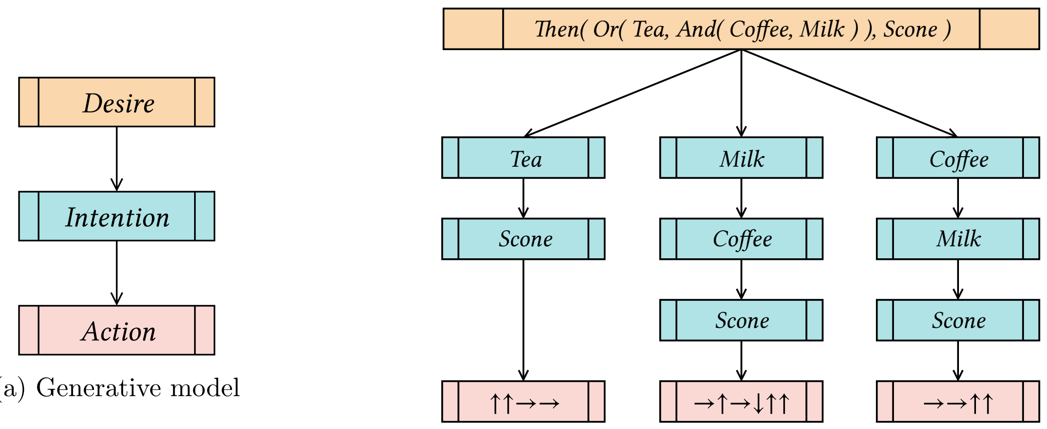
1.1 Desires

We routinely act in the world. And generally, we're not random actors. But rather actors directed at what we want, our desires. As observers we also assume this about others, and use the way others act in order to infer what they want [1].

Try to infer what I want for breakfast. I have coffee, tea, scones, and milk. To start, I brew and pour myself a cup of tea. At this point it's clear I at least want tea. Yet you also learn what I don't want. I don't only want coffee, or scones, or milk. My cup of tea contradicts that. And if I did want those things after my tea, having them first isn't critical. In other words, I may want tea first and then a scone. But I definitely don't want the reverse.

After my tea I grab a scone. I quickly eat it and head to work. So at first glance, there is no mystery. I want tea first and then a scone. It's a good inference, but not the only one. Even though I brew tea first, it's possible I want coffee with milk instead. Maybe I was late for work, and my coffee takes more time to prepare than tea. If I enjoy both options, I choose the option which saves me time. In this case what I really want is first tea, or coffee and milk. Whichever is more convenient. And after, a scone.

Desires are complex. We factor not only what actions fulfill them, but also the cost and reward for different ways to fulfill the same desire [7]. Moreover, desires compose. In this work, we focus on three ways that they do. Desires compose for order (wanting tea first, *then* a scone). Desires compose for choice (wanting tea *or* coffee). And desires compose for choice and order (wanting tea



(a) Generative model

(b) Example of the generative process

Figure 1-1: Sketch of the general model, with an example of how desire generates intention which generates action. In (b), the expression corresponds to an agent who first wants either coffee and milk, or just tea followed by a scone afterwards. Note that the actions are the cardinal directions, which reflect a modelling choice we explain in Chapter 2. In our setup, one satisfies intentions and desires by navigating a gridworld environment. For example, moving to a grid occupied by tea amounts to acquiring tea.

and a scone, in any order).

Figure 1-1 sketches our view of desires, and a way to make sense of my breakfast choice. Desire describes what we want. Intention describes what choices and orderings matter to fulfill my desire. Action describes what we do in the world to fulfill those intentions, and in turn our desires. So if my desire is to have tea and a scone in any order, two intentions spur. Either I have tea first and then a scone, or a scone first and then tea. From both intentions spur distinct actions which fulfill them.

This view both complements and contrasts previous work on how humans act on and infer desires. To complement, we assume that humans act on their desires in ways which maximize utility—the reward of action minus its cost [7]. We also take a Bayesian lens on how humans infer desires. We assume that when humans observe action, they have prior beliefs about the possible desires one could have. And that they integrate these prior beliefs into the likelihood of different desires [1, 2].

To contrast, we discuss desires which compose to express order and choice. Previous works focus on how humans infer simpler desires [1, 2]. These are desires where you only want a single thing, like tea. This work subsumes and replicates those works. Yet we provide novel evidence and modelling techniques for how humans act on and infer a far broader class of desires. These include simpler desires and those which compose to express complex combinations of order and choice like in Figure 1-1b.

These kinds of desires constrain our modelling choice. They must compose for order and choice. And they must confer predictions about how humans act on and infer their desires. Insofar as these predictions match those of humans, our modelling choice gains empirical weight. Lambda calculi meet these needs. We now show why they make a precise modelling choice.

1.2 Programs

Lambda calculi are models of computation akin to Turing machines [3]. They are sets of rules which build and execute lambda terms (programs). These rules specify programming languages: what their programs look like (syntax) and how they behave (semantics). Because these rules are formal, lambda calculi excel as tools to build and reason about programs. They are the lingua franca of programming language research [10]. By tweaks to the rules of a lambda calculus, you create a new programming language.

Let's look at a program which computes identity, $f(x) = x$.

$$(\lambda x.x)$$

$(\lambda x.x)$ is a program. x is its input. x is its body. The body takes the input and returns an output. If we apply the program to an argument, computation unfolds.

$$(\lambda x.x)1 \quad \rightarrow \quad x[1\backslash x] \quad \rightarrow \quad 1$$

First, we apply the program to an argument, $(\lambda x.x)1$. Then we substitute the argument into the body of the program, $x[1\backslash x]$. The computation halts and returns 1.

$\rightarrow$ is a relation, one we define with a set of rules. It defines an operational semantics for programming languages [10]. It tells us how the language operates, how the programs execute, how they behave. From it you can prove properties about your programming language like whether it's Turing complete [3]. Moreover, operational semantics act as blueprints for the implementation of programming languages. They're succinct descriptors which facilitate the dissemination and replication of programming languages.

I reserve the formal details for later. But to convey intuition, here are ways we will model desires from Figure 1-1 using lambda calculi.

$$(\textsc{Then}\ x\ y) \quad\rightarrow\quad (\textsc{Then}\ [x]\ [y]) \quad\rightarrow\quad [x, y] \quad\rightarrow\quad [\uparrow, \uparrow, \rightarrow, \rightarrow]$$

($\textsc{Then}$ x y) is a program. $\textsc{Then}$ is a function over desires, x and y, which encodes order. We encode order using lists. So over the course of the computation, the program ends up building an ordered list, $[x, y]$, which means I want x first, then y. Think of the ordered list as how we construe intentions. It encodes the order which matters to fulfill our desire. After we know the constraints on order, we can then act to fulfill the desire $[\uparrow, \uparrow, \rightarrow, \rightarrow]$. This is a brief glimpse of the operational semantics of the programming language we build and use in this work.

Yet another virtue of lambda calculi are how they help us clarify compositionality [12]. So far we see that desires compose to express complex combinations of order and choice. Further, we see ways in which programs can model these desires. But are they compositional, and in what sense?

1.3 Compositionality

The principle of compositionality states that the meaning of an expression is a function of the meaning of its parts [12]. It entails specific behaviors. For example, suppose you know the meaning of ($\textsc{Then}$ x y). If compositionality holds, then it must be that you know the meaning of $\textsc{Then}$, x, and y individually. And as a result of knowing those meanings, you should also know the meaning of:

$$(\textsc{Then}\ x\ (\textsc{Then}\ y\ x)) \quad (\textsc{Then}\ y\ (\textsc{Then}\ x\ y)) \quad (\textsc{Then}\ x\ (\textsc{Then}\ y\ (\textsc{Then}\ x\ y))) \quad \cdots$$

They all are made of the same parts. With compositionality, knowing the meanings of those few parts lets you know the meaning of all their combinations.

Many fields find compositionality useful on similar merits: philosophy, linguistics, logic, artificial intelligence, and programming languages. Yet its exact usage varies considerably, in part because the general principle makes no commitment as to what counts as an expression, a function, or a meaning [12].

To make our claims of compositionality precise, we will abide by its usage in programming languages. There are two key reasons. First, its usage is not tenuous in programming languages. Second, our modelling choice in lambda calculi make this usage a natural choice. For decades, work on denotational semantics of lambda calculi provide a formal framework for making claims about compositionality [8]. It makes compositionality a mathematical criteria.

Denotational semantics assign denotations (meanings) to arbitrary programs in lambda calculi. These denotations are mappings from the program to some kind of mathematics. For example, a simple denotational semantics maps programs to sets and the functions between them. All possible inputs to a program denote a set. All possible outputs to a program denote a set. Programs with those inputs and outputs denote a function between those sets. To check compositionality, you check if the denotation is compositional. Since the denotation exists in some kind of mathematics, it's only a matter of checking whether the denotation of the whole is literally a composition of the denotation of the parts.

As we segue into the formal specification of our model, keep in mind the merits of a little bureaucracy of notation. We facilitate replication, in the same vein that lambda calculi do in programming language research. And we make precise the usage of terms whose usage can vary in unhelpful ways: program, compositionality, and meaning.

Chapter 2

Act, a programming language

Programming languages let us specify a space of programs. Using lambda calculi, we can control how they look (syntax) and how they behave (semantics). In this section we draw on tools from programming language research to specify Act, a programming language whose programs are desires. As these programs execute, they evaluate to actions. Later, we show that programs in Act predict how humans act on and infer desires.

2.1 Syntax

Act has minimal syntax. It's much simpler than Python, MATLAB, or any other programming language of choice. Figure 2-1 specifies this syntax in Backus-Naur form. This form lets you recursively determine the structure of your programming language. In Act, there are three key structures: desires, intentions, and actions. As programs execute, desires become intentions, and intentions become actions. We now illustrate what these structures look like in Act.

Desires are represented in the syntax as d. Any valid expression you derive for d counts as a program. There are three kinds of entities which make up d: individual objects, choices between objects, and orderings between objects. Let's derive a few desires from the syntax to make this clear:

$$\begin{aligned} d &::= \\ &\quad o_3 \end{aligned}$$

$$\begin{aligned} d &::= \\ &\quad (\textsc{Or}\ d\ d) \\ &\quad (\textsc{Or}\ o_1\ o_2) \end{aligned}$$

$$\begin{aligned} d &::= \\ &\quad (\textsc{Then}\ d\ d) \\ &\quad (\textsc{Then}\ o_1\ (\textsc{Or}\ d\ d)) \\ &\quad (\textsc{Then}\ o_1\ (\textsc{Or}\ o_5\ o_3)) \end{aligned}$$

Going left to right, the first desire is o_3. We interpret this desire as one wanting a single object o_3. The second program is ($\textsc{Or}$ o_1 o_2). We interpret this desire as one wanting either the object o_1 or the object o_2. The third desire is ($\textsc{Then}$ o_1 ($\textsc{Or}$ o_5 o_3)). We interpret this desire as one first wanting the object o_1, and then either objects o_5 or o_3.

To derive more desires from the syntax, you pick a valid expression from d, and then repeat until you're left without expressions to fill for d. Note how if you pick ($\textsc{Or}$ d d) it will require you to pick two other expressions for d. We illustrate the process line by line in the previous example derivations.

SYNTAX

$d ::=$	DESIRES:
$o_1 \mid o_2 \mid \cdots \mid o_n$	*objects*
$(\textsc{Or}\ d\ d)$	*choice*
$(\textsc{Then}\ d\ d)$	*order*
i	*intention*
$i ::=$	INTENTIONS:
$[o_1] \mid [o_2] \mid \cdots \mid [o_n]$	*objects*
$[i, i]$	*choice*
$i \times i$	*order*
$a ::=$	ACTIONS:
$[\rightarrow] \mid [\uparrow] \mid [\leftarrow] \mid [\downarrow]$	*single*
$[\nearrow] \mid [\nwarrow] \mid [\swarrow] \mid [\searrow]$	
$a \times a$	*multiple*

SEMANTICS $\boxed{d \rightarrow a}$

$$\frac{d_1 \rightarrow d_1'}{(*\ d_1\ d_2) \rightarrow (*\ d_1'\ d_2)}\ (\textsc{Arg-l})$$

$$\frac{d_2 \rightarrow d_2'}{(*\ d_1\ d_2) \rightarrow (*\ d_1\ d_2')}\ (\textsc{Arg-r})$$

$$\frac{}{o_i \rightarrow [o_i] : 0 \leq i \leq n}\ (\textsc{Obj-list})$$

$$\frac{}{(\textsc{Or}\ i_1\ i_2) \rightarrow [i_1, i_2]}\ (\textsc{Or-list})$$

$$\frac{}{(\textsc{Then}\ i_1\ i_2) \rightarrow i_1 \times i_2}\ (\textsc{Then-list})$$

$$\frac{}{i \xrightarrow{\mathcal{M}} a}\ (\textsc{Mdp-act})$$

Figure 2-1: Syntax and semantics of ACT

Intentions are represented in the syntax as i. Essentially, they're lists of objects. Each list describes an ordering of objects. And with a list of lists, each list describes choices between orderings on objects. There are three kinds of lists which make up i: a one-element list containing an object, a list of lists, and the cartesian product of two lists. Let's derive a few intentions from the syntax to make this clear:

$$\begin{array}{lll}
i ::= & i ::= & i ::= \\
\quad [o_3] & \quad [i, i] & \quad [i, i] \\
 & \quad [[o_1], [o_2]] & \quad [(i \times i), (i \times i)] \\
 & & \quad [[o_1, o_5], [o_1, o_3]]
\end{array}$$

Going left to right, the first intention is $[o_3]$. We interpret this intention as a list with one object o_3, equivalent to wanting only object o_3. The second intention is $[[o_1], [o_2]]$. We interpret this intention as a list of lists, one list with object o_1 and another list with object o_3. Each list describes something one could want, so together the list of lists describes wanting either objects o_1 or o_3. The third intention is $[[o_1, o_5], [o_1, o_3]]$. We interpret this intention as a list of lists again, but now each list has two elements. The first list describes an ordering of objects, wanting first object o_1 and then object o_5. Likewise, the second list describes wanting first object o_1 and then object o_5. Together, the list of lists describes a choice between these orderings.

To derive more intentions from the syntax, use the same strategy from d. The only difference to note is the cartesian product $\times$. With simple lists, it joins them together i.e. $[o_1, o_2] \times [o_3] = [o_1, o_2, o_3]$. With lists of lists, the cartesian product has more nuance, i.e. $[[o_1, o_2], [o_3, o_4]] \times [o_5] = [[o_1, o_2, o_5], [o_3, o_4, o_5]]$. In general, its behavior mimics the set-theoretic definition where $+$ is a list and string concatenation operator:

$$A \times B = \{[a + b] \mid a \in A \text{ and } b \in B\}$$

Actions are represented in the syntax as a. They're the simplest forms in ACT. They're just lists of actions, where actions are arrows in the cardinal directions. Any list of arrows counts as an action. For good measure, we derive a few actions from the syntax to make this clear:

$a ::=$

$[\rightarrow]$

$a ::=$

$[a_1 + a_2]$

$[\rightarrow, \rightarrow]$

$a ::=$

$[a + a]$

$[(a + a) + (a + a)]$

$[\rightarrow, \rightarrow, \uparrow, \uparrow]$

Going left to right, the first action is $[\rightarrow]$. We interpret this action as taking one step to the right. The second action is $[\rightarrow, \rightarrow]$. We interpret this action as taking two steps to the right. The third action is $[\rightarrow, \rightarrow, \uparrow, \uparrow]$. We interpret this action as taking two steps to the right, then two steps up.

These syntactic forms for d, i, and a are all there is to what programs look like in Act. Yet it's still unclear how these programs behave: how desires become intentions and how intentions become actions. To these questions we introduce the semantics of Act.

2.2 Semantics

There are several ways to give semantics to a programming language [13]. Operational semantics is one kind. These tell us how programs execute, how they operate. The meaning of a program comes in virtue of its operation. Operational semantics are especially useful in providing a blueprint for implementation.

Denotational semantics is another kind. These tell us how programs map to mathematical objects, what their denotations are. The meaning of a program comes in virtue of its denotation in some mathematics. Denotational semantics are especially useful in clarifying in what sense something is compositional, as these semantics make compositionality a mathematical criteria.

2.2.1 Operational

Earlier we saw what desires, intentions, and actions look like in Act. Now we cover how desires become intentions, and intentions become actions.

In Figure 2-1, the operational semantics are given by a set of inference rules [10]. Altogether these rules formally specify a relation $d \rightarrow a$, which describes how programs execute in Act. Each rule has a simple underlying logic. When the premise above the line holds, you may conclude what is below the line. In the context of Act, a premise is a program and the conclusion is a step of computation the program can take. As a result, these rules let us know precisely how a program may compute. Each step in the computation is described by the use of a rule. Let's run through each rule, and then illustrate their use with a few examples.

(Obj-list) is an inference rule which tells us how desires which are single objects become intentions. Because the desire is merely wanting one object, the intention is a list containing just that object. Since this rule requires no premise above the line, you can always use it when a program is in the form o_i for $0 \leq i \leq n$. Here's an example of how a simple desire like o_3 becomes an intention $[o_3]$.

$$\frac{}{o_3 \rightarrow [o_3]} \text{ (Obj-list)}$$

(Or-list) is an inference rule which tells us how desires which express choice become intentions. We encode choice as a list of lists, where each list represents a possible intention to act on. Since this rule requires no premise above the line, you can always use it when a program is in the form (Or i_1 i_2). Here's an example of how a desire expressing choice like (Or $[o_1, o_2]$ $[o_3, o_4]$) becomes an intention $[[o_1, o_2], [o_3, o_4]]$.

$$\frac{}{(\text{Or } [o_1, o_2]\ [o_3, o_4]) \rightarrow [[o_1, o_2], [o_3, o_4]]} \text{ (Or-list)}$$

(Then-list) is an inference rule which tells us how desires which express order become intentions. We encode order in a list. The first element is what you want first, the second element is what you want second. Since this rule requires no premise above the line, you can always use it when a program is in the form (Then i_1 i_2). Here's an example of how a desire expressing order like (Then $[[o_1], [o_2]]$ $[[o_3], [o_4]]$) becomes an intention $[[o_1, o_3], [o_1, o_4], [o_2, o_3], [o_2, o_4]]$.

$$\frac{}{(\text{Then } [[o_1], [o_2]]\ [[o_3], [o_4]]) \rightarrow [[o_1, o_3], [o_1, o_4], [o_2, o_3], [o_2, o_4]]} \text{ (Or-list)}$$

(Arg-l) is an inference rule which tells us how to compute on the left argument d_1 of either (Or d_1 d_2) or (Then d_1 d_2). The premise of the rule requires that d_1 compute to some value. If it does, then you may replace d_1 with the value it computes to. Here's an example where we know $o_1 \rightarrow [o_1]$, hence (Or o_1 o_2) $\rightarrow$ (Or $[o_1]$ o_2).

$$\frac{o_1 \to [o_1]}{(\textsc{Or}\ o_1\ o_2) \to (\textsc{Or}\ [o_1]\ o_2)}\ (\textsc{Arg-l})$$

(Arg-r) is an inference rule which tells us how to compute on the right argument d_2 of either (Or d_1 d_2) or (Then d_1 d_2). It's essentially the same as (Arg-l). The premise of the rule requires that d_2 compute to some value. If it does, then you may replace d_2 with the value it computes to. For good measure, here's an example where we know (Then $[o_1]$ $[o_2]$) $\to [o_1, o_2]$, hence (Then o_3 (Then $[o_1]$ $[o_2]$)) $\to$ (Then o_3 $[o_1, o_2]$).

$$\frac{(\textsc{Then}\ [o_1]\ [o_2]) \to [o_1, o_2]}{(\textsc{Then}\ o_3\ (\textsc{Then}\ [o_1]\ [o_2])) \to (\textsc{Then}\ o_3\ [o_1, o_2])}\ (\textsc{Arg-r})$$

(Mdp-act) is an inference rule which tells us how intentions become actions. It merits extra discussion. It passes intentions to a Markov Decision Process (MDP), which probabilistically determines which actions to take in the world according to the given intent. MDPs are standard tools in the robotics and planning research [11]. Previous works also leverage MDPs to predict how humans act on and infer their desires [6, 1, 2]. Importantly, the addition of this rule makes Act a probabilistic programming language. Programs don't always return the same output. Rather, they return outputs according to some distribution.

For a fuller description of MDPs I defer to the standard texts [11]. But to convey intuition, let's discuss a bit of their workings and how we translate intentions to actions via MDPs. An MDP is a 4-tuple $\mathcal{M} = (S, A, T, R)$, where

(i) S is a set of states. We might carve up a map into squares, calling each square a state. Being in one of those squares amounts to being in some state of the world.

(ii) A is a set of actions. If our states are squares in a map, then our actions could be the cardinal directions. The action $\to$ moves an agent one square to the right.

(iii) $T(s, a, s')$ is a transition function. It tells us the probability of an action a taken at state s successfully moving an agent to state s'. Often actions in the world aren't deterministic, and T helps to capture this reality.

(iv) $R(s)$ is a reward function. It tells us the reward an agent gets when in state s of the world. Imagine the object o_3 is in some state of the world s_{o_3}, some square in a map. If my desire is to have o_3, then $R(s_{o_3}) > 0$. For all other states s, $R(s) = 0$.

When we define those sets and functions, we have an MDP. Given an MDP, there exist a number of algorithms which solve them. A solution to an MDP finds the courses of action from any state which maximizes expected reward to an agent [11]. So if my desire is to get an object o_1, an MDP can tell me what to do from any state of the world. Likewise, if my desire is to get an object o_2 then another MDP can tell me what to do from any state of the world.

If you construct n MDPs for the n objects one could desire, you in turn get complete information about the course of actions to satisfy desires with arbitrary constraints on order and choice. Later we discuss how that information helps us construct distributions over desires, intentions, and actions. For now, just note that there generally exists an MDP for which (Mdp-act) holds:

$$\frac{}{[o_1, o_2] \xrightarrow{\mathcal{M}} [\uparrow, \uparrow, \rightarrow, \rightarrow]}\ (\textsc{Mdp-act})$$

All these rules constitute the operational semantics. If you use them together, it's possible to construct a proof of how programs execute in Act. In the proof which follows we assume there exists an MDP for which the application of (Mdp-act) holds.

$$\begin{aligned}
&(\textsc{Or}\ (\textsc{Then}\ o_1\ o_2)\ (\textsc{Then}\ o_3\ o_4))\\
\rightarrow\ &(\textsc{Or}\ (\textsc{Then}\ [o_1]\ o_2)\ (\textsc{Then}\ o_3\ o_4))\\
\rightarrow\ &(\textsc{Or}\ (\textsc{Then}\ [o_1]\ [o_2])\ (\textsc{Then}\ o_3\ o_4))\\
\rightarrow\ &(\textsc{Or}\ (\textsc{Then}\ [o_1]\ [o_2])\ (\textsc{Then}\ [o_3]\ o_4))\\
\rightarrow\ &(\textsc{Or}\ (\textsc{Then}\ [o_1]\ [o_2])\ (\textsc{Then}\ [o_3]\ [o_4]))\\
\rightarrow\ &(\textsc{Or}\ [o_1, o_2]\ [o_3, o_4])\\
\rightarrow\ &[[o_1, o_2], [o_3, o_4]]\\
\xrightarrow{\mathcal{M}}\ &[\uparrow, \uparrow, \rightarrow, \rightarrow]
\end{aligned}$$

In addition to clarifying execution, an operational semantics lets us freely create derived forms [10]. It's sort of like defining new functions in your programming language by using old ones. In Act we do this with the And operator. Sometimes you may want o_1 and o_2, in any order. This desire combines constraints on order and choice, and is common enough that we ought to have a form in the language for it. But since it's possible to define this operator in terms of $\textsc{Or}$ and $\textsc{Then}$ there's no need to wire it in our semantics. We instead state the derived form here.

$$\frac{}{(\textsc{And}\ d_1\ d_2) \to (\textsc{Or}\ (\textsc{Then}\ d_1\ d_2)\ (\textsc{Then}\ d_2\ d_1))}$$

2.2.2 Denotational

Operational semantics state how computation unfolds. In $\textsc{Act}$, this describes how desires become intentions, and intentions become actions. However, these semantics do not state how programs are compositional. For this, we need denotational semantics.

Suppose we construe denotations (meanings) as the actions which a desire (program) produce. For these denotations to be compositional, these actions must be a composition of the actions which follow from the program's parts. We now show that these denotations can't be compositional in general. In what follows, we mark denotations with double bracket notation $[\![d]\!]$

Imagine you are in a square room, in the bottom left corner. You first want object o_1 in the top left corner. Then you want object o_2 in the top right corner. We can build an MDP where the denotation of your desire is $[\![(\textsc{Then}\ o_1\ o_2)]\!] = [\uparrow, \uparrow, \rightarrow, \rightarrow]$. These actions take you to o_1 and then to o_2. But are these actions compositional? Let's check. If you only wanted o_1 then you would go straight to the top left corner, given by $[\![o_1]\!] = [\uparrow, \uparrow]$. If you only wanted o_2 then you would go straight to the top right corner, given by $[\![o_2]\!] = [\nearrow, \nearrow]$. Recall that $\textsc{Then}$ acts as the cartesian product operator for lists, so $[\![\textsc{Then}]\!] = \times$. With the denotations of each part in $(\textsc{Then}\ o_1\ o_2)$, it's clear that they don't compose in any way to produce the denotation of the whole expression $[\![(\textsc{Then}\ o_1\ o_2)]\!] = [\uparrow, \uparrow, \rightarrow, \rightarrow]$. Hence these denotations aren't compositional, desires aren't compositional with respect to actions.

It turns out that a slight adjustment to the operational semantics recover compositionality. If we take the operational semantics minus the ($\textsc{Mdp-act}$) rule, we can compute a list of intentions for an arbitrary program $d \to i$. Let's construe denotations as the intentions which a desire (program) produce. Now we can replay the previous situation and see if these denotations are compositional.

Imagine you are in a square room, in the bottom left corner. You first want object o_1 in the top left corner. Then you want object o_2 in the top right corner. It's always the case that the denotation of this desire is $[\![(\textsc{Then}\ o_1\ o_2)]\!] = [o_1, o_2]$. This intention encodes the constraint on order necessary to fulfill the desire: you want o_1 first, and then o_2. But is this intention compositional? Let's check. If you only want o_1 then its corresponding intention is given by $[\![o_1]\!] = [o_1]$. If you only want o_2 then its corresponding intention is given by $[\![o_2]\!] = [o_2]$. Recall that Then acts as the cartesian product operator for lists, so $[\![\textsc{Then}]\!] = \times$. With the denotations of each part in $(\textsc{Then}\ o_1\ o_2)$, it's clear that they do compose to produce the denotation of the whole expression $([o_1]\ \times\ [o_2]) = [\![(\textsc{Then}\ o_1\ o_2)]\!] = [o_1, o_2]$. This is true in general for arbitrary desires in Act, their denotations are always compositional with respect to intentions.

This is useful to know. With knowledge of just the denotations from the previous example you can interpret all potential orderings on objects o_1 and o_2:

$$[\![(\textsc{Then}\ o_1\ (\textsc{Then}\ o_1\ o_2))]\!] \quad [\![(\textsc{Then}\ o_1\ (\textsc{Then}\ o_2\ (\textsc{Then}\ o_1\ o_2)))]\!] \quad \cdots$$

What makes compositionality useful is this productivity. Few denotations give you plenty power to reason about the behavior of your programs. When denotations aren't compositional, this doesn't work. To reason about the behavior of your programs requires more than just knowledge of the denotation of its parts.

2.3 To act on and infer desires

The key claim of this work is that Act predicts how humans act on and infer desires. And earlier we said that Act is actually a probabilistic programming language because of (Mdp-act). As such, Act entails probability distributions over desires, intentions, and actions. In this section, we characterize those distributions. In experiments to follow, we show that humans act on and infer desires in close agreement to these distributions.

2.3.1 Actions given desires

Let's start with acting on desires. The following distribution succinctly describes the probability of action given some desire. However there is nuance to each piece of the equation, which merit discussion.

$$P(a|d) = \sum_i P(a|i)P(i|d)$$

We assume people satisfy their desires in ways which maximize utility, which balance the cost and reward of action. If what I want is across the street, I'm likely to take the direct path there. $P(a|i)$ captures this. Likewise, we assume people consider the utility of different intentions which satisfy a desire. If I can either go across the street or to a different country to get what I want, I'm likely to just cross the street. Assuming the reward is the same for either intention, I choose the intention with lower cost. $P(i|d)$ captures this.

Rewards and costs are explicit in an MDP, and so it's possible to extract the utility of actions and intentions to compute these distributions. We denote utility as follows:

$$U(x) = \frac{R(x) - C(x)}{\tau}$$

where x is an intention or action, R is a reward function, C is a cost function, and τ is a parameter which controls how much minimizing the costs of intentions or actions impact the overall utility. As τ grows, so do the utility of suboptimal actions or intentions. But U does not return a probability distribution, so we use a softmax function to convert utility of different actions and intentions into distributions for both.

$$P(a|i) = \frac{e^{U(a)}}{\sum\limits_a e^{U(a)}} \qquad\qquad P(i|d) = \frac{e^{U(i)}}{\sum\limits_i e^{U(i)}}$$

With the softmax over utilities, we have all the machinery necessary to compute $P(a|d)$, the probability of action given some desire. We end up reusing this machinery for the inference problem too—by applying Bayes' rule.

2.3.2 Desires given action

If you see someone enter a grocery store, you might infer they want groceries. But you might also infer they want to use the bathroom. Both are consistent to the evidence, seeing someone enter the grocery store. Yet one seems more likely. To infer desires, we need a way to integrate our observations of action with our prior conceptions about reasonable desires. Bayes' rule provides an elegant way to do this.

$$P(d|a) \propto P(a|d)P(d)$$

Our likelihood $P(a|d)$ captures how consistent the data are to some hypothesis. In our case the data are actions and hypotheses are desires. Our prior $P(d)$ captures, without seeing any evidence, how likely different desires are. We already know how to construct the likelihood $P(a|d)$, so all we need is a sensible way to construct the prior.

In ACT, we decide that more complex programs should be a priori less likely candidates for desires. We determine complexity by the number of times a rule from Figure 2-1 is used to derive the expression.

d ::=

o_3

d ::=

(OR d d)

(OR o_1 d)

(OR o_1 o_2)

d ::=

(THEN d d)

(THEN d (OR d d))

(THEN o_1 (OR d d))

(THEN o_1 (OR o_5 d))

(THEN o_1 (OR o_5 o_3))

Going left to right, the complexity of o_3 is 1. The complexity of (OR o_1 o_2) is 3. The complexity of (THEN o_1 (OR o_5 o_3)) is 5. Let's call $O(d)$ a function which returns a desire's complexity. Then we can generate our prior distribution over desires using a standard exponential distribution.

$$P(d) = \lambda e^{-\lambda O(d)}$$

Because much of the probability mass in these distributions are focused on desires of smaller complexity, we do inference by enumeration up to desires of a certain complexity. Though it's possible to use other sampling-based approaches as well.

Chapter 3

Experiments

3.1 Experiment no. 1

3.1.1 Design

We want to know if compositional programs in Act predict the way humans act on and infer desires. So we design an inference task where humans observe an agent's behavior across one or two days. Afterwards, they rate how much they think the agent had different desires. From these ratings we recover an approximate distribution to $P(d|a)$ for the desires, intentions, and actions we experimentally test. From it, we can compare to $P(d|a)$ from Act. Recall that $P(d|a)$ uses how humans act on desires in the likelihood $P(a|d)$. As a result, similarities in the empirical $P(d|a)$ and Act's $P(d|a)$ confer predictions about how humans act on and infer desires.

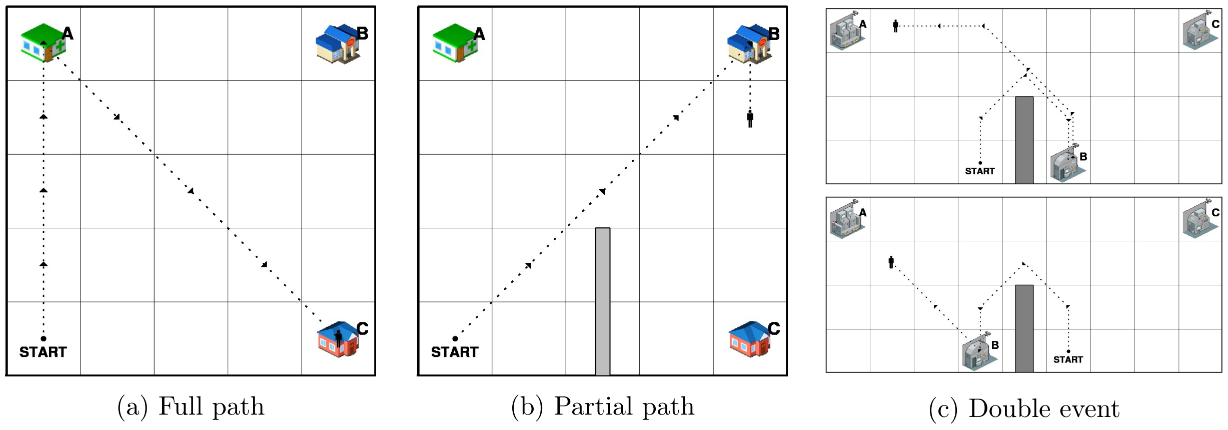
3.1.2 Methods

Participants

33 participants recruited using Amazon's Mechanical Turk Framework.

Stimuli

Figure 3-1a shows an example of the stimuli. They consist of 19 two-dimensional images of an agent who travels to one or more of three potential locations. Eight of these trials are a single event, the remaining 11 consist of two events. We avoid cherry picking convenient stimuli by parametrically



(a) Full path (b) Partial path (c) Double event

Figure 3-1: Environments where an agent can have compositional desires. These are also the variants we use as stimuli in our experiments. (a) is from Experiment 1. (b) is from Experiment 2. (c) is from Experiment 3.

generating a set of stimuli to test.

To construct the single event trials, we enumerate all possible efficient paths an agent could take to reach between one and three of the locations. We then filter this set by removing paths which were equivalent under rotation or reflection of the map.

To construct double event trials, we enumerate all possible efficient paths between one and two of the locations, omitting paths between three locations to prevent the set from growing too large. Unlike the single event trials, we don't remove paths equivalent under rotation or reflection. We require these paths to construct the most primitive desires which occur over two events, e.g. (Or o_1 o_2). This creates a base set of nine paths.

We then split these nine paths into two classes, paths which go to one object and paths which go to two objects. For each class we compute the cartesian product of itself, removing duplicate pairs of stimuli (e.g., the pair $(o_1, o_2) = (o_2, o_1)$). Note that the pair denotes a double event, where each element is an observation on one day. Under a few criteria, we can further pare down the size of the set so it's experimentally tractable.

If pairs violate the principle of rational action, we remove them. For example, when on one day the agent travels to the furthest object from them (so they want that object), and on the second day they grab a different object which is closer to them. Because the map is constant across both days and we assume the agents desire persist across both days, then grabbing the different object on the second day violates rational action.

Other criteria include whether a pair is the reflection or rotation of another pair, e.g. between (o_1, o_1) and (o_3, o_3) we only keep one. And lastly, we remove pairs which only have one plausible desire which explain them. These impact our ability to get graded responses from participants. For example if on both days the agent travels to the furthest object, then it's clear they just want that object. This example won't confer graded judgments from participants, nor from Act. However, as an exception we include one of these cases just to show that Act predicts this outcome.

From those criteria there end up being 11 double event stimuli and 8 single event stimuli. In total, these comprise the 19 stimuli we use in the experiment.

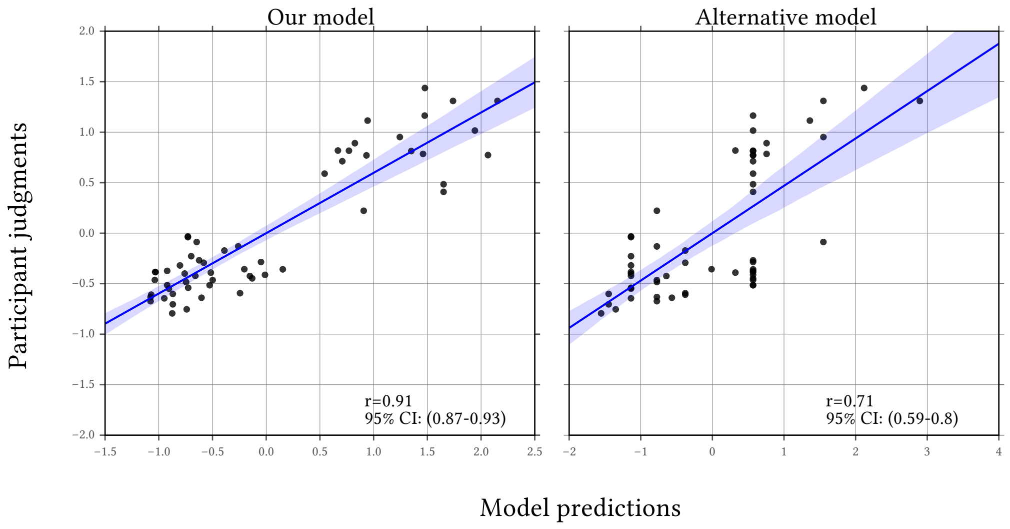


Figure 3-2: Comparison between Act and the alternative model. X-axis are ratings given by models. Y-axis are ratings given by humans. We z-score ratings for comparison.

Procedure

Participants first read a tutorial which explains the logic of the task. They then complete a survey afterwards to ensure they read the instructions. If they pass, we test them on the task.

During the task, participants complete 19 trials. In each trial participants see a stimulus on the left half of their screen. The stimulus consists of an agent's path toward different objects in a small map, as in Figure 3-1a. On the right half of their screen we ask them to rate their belief for three candidate desires. The ratings scale from 0-10 for each desire. 0 indicates "definitely not", 5 indicates "maybe", and 10 indicates "definitely".

To avoid cherry picking candidate desires which would favor Act, we generate candidate desires with the following procedure. For each trial, normalize the posterior distribution $P(d|a)$ for both Act and the alternative model (which we discuss next). Then for each trial, construct a set containing the top three desires with highest probability for both Act and the alternative model. If on any trial these sets contain duplicate desires, keep the one with the highest probability. Finally, for each trial extract the top three desires. These are the candidate desires we show to participants. With this procedure we avoid choosing desires which strictly favor what Act finds plausible.

Since the candidate desires from Act lack interpretability under appropriate expertise, we translate them into natural language descriptions for our participants. For example, (Or o_1 o_2) would be "the agent wants either o_1 or o_2". To ensure these translations are accurate, two coders translate the natural language descriptions back to their programmatic form in Act—despite being blind to their original form. These coders show full agreement and correctly translate back to the programmatic form in Act for all trials.

Alternative model

A simple alternative model to Act is another programming language with the same syntax but a simpler semantics. Currently, the semantics of Act involve the use of an MDP which let us treat humans as maximizing utility. These ideas are borne out in $P(a|d)$, which is a key piece of machinery we use to characterize how people act on and infer desires.

$$P(d|a) \propto P(a|d)P(d)$$

In our alternative model, we do away with the assumption of humans as utility maximizers. We test a simpler, deterministic likelihood.

$$P(a|d) = \begin{cases} 1, & \text{if } a \text{ satisfies } d \\ 0, & \text{otherwise} \end{cases}$$

Yet we pair this simpler likelihood with the same prior $P(d)$ which penalizes more complex desires when making inferences.

3.1.3 Results

Figure 3-2 provides a summary of our results, comparing both Act and the alternative model to human predictions on the desire inference task. Act fits human judgments well, showing a correlation of $r = .91$ with participant judgments (95% CI: $.87 - .93$).

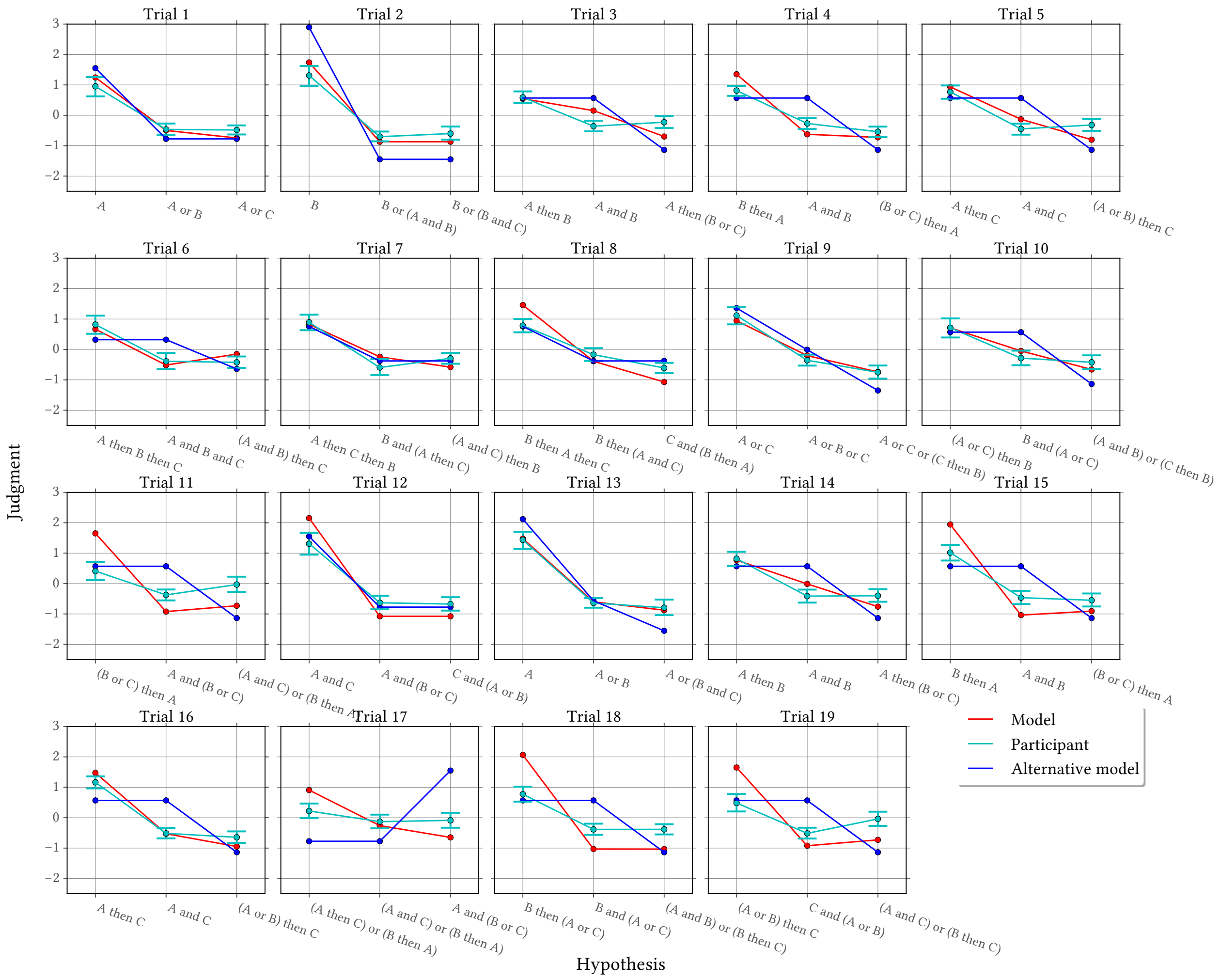


Figure 3-3: Trial by trial comparison between our model, alternative model, and humans on the inference task. X-axis are the candidate desires. Y-axis are the ratings. We z-score ratings for comparison.

For a finer grain look at the results, we show the trial by trial comparison as well in Figure 3-3. For each trial you can see where humans diverge or converge with Act and the alternative model. Qualitatively, you see nice convergence between humans and Act across the trials. On many trials the alternative similarly converges with a few exceptions which hurt its overall fit. We take a closer look at why in Figure 3-4, which are the results from the stimulus in Figure 3-1a.

In trial 5, the agent goes to A and then to C. So a sensible response is that they want (THEN A C). Another sensible inference is that the agent merely wants both in any order, (AND A C). But since the evidence only shows one ordering, that possibility is less likely. Both humans and ACT capture this, yet the alternative model does not. This is a result of the alternative model's likelihood, which highly rates actions which satisfy a desire irrespective of considerations to utility.

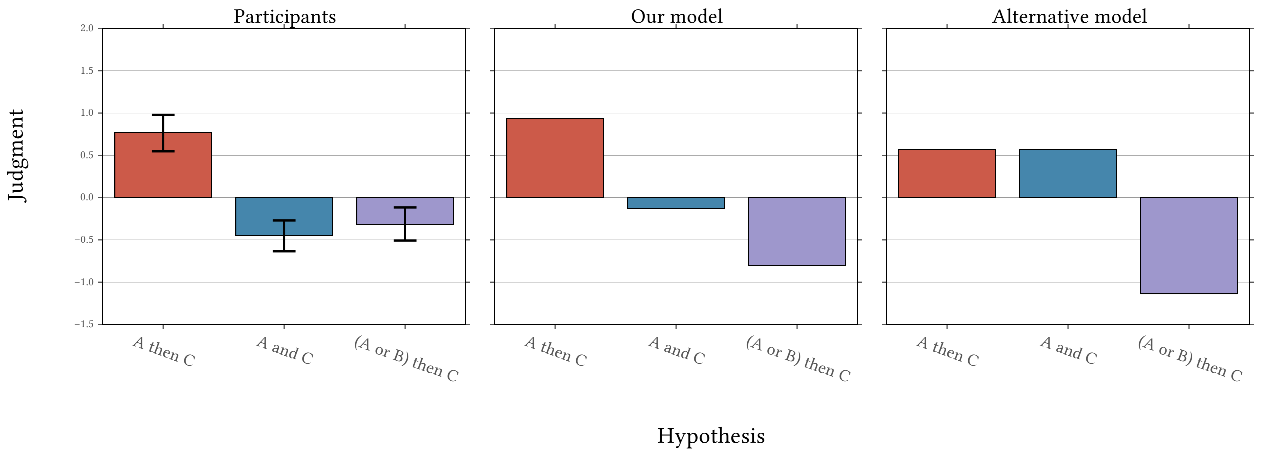


Figure 3-4: In depth look at trial 5. X-axis are candidate desires. Y-axis are the ratings. We z-score ratings for comparison.

3.2 Experiment no. 2

3.2.1 Design

We want to know if compositional programs in ACT predict the way humans act on and infer desires. So we design an inference task similar to experiment no. 1 where humans observe an agent's behavior across one or two days. The distinction is that now the agent's behavior is partial, they do not finish reaching their goal. Afterwards, they rate how much they think the agent had different desires. From these ratings we recover an approximate distribution to $P(d|a)$ for the desires, intentions, and actions we experimentally test. From it, we can compare to $P(d|a)$ from ACT. Recall that $P(d|a)$ uses how humans act on desires in the likelihood $P(a|d)$. As a result, similarities in the empirical $P(d|a)$ and ACT's $P(d|a)$ confer predictions about how humans act on and infer desires.

3.2.2 Methods

Participants

30 participants recruited using Amazon's Mechanical Turk Framework.

Stimuli

Figure 3-1b shows an example of the stimuli. They consist of 16 two-dimensional images of an agent who travels to one or more of three potential locations. Eight of these trials have a wall in the middle of the environment, which restricts potential paths for the agent. The other eight have no wall in the environment meant to block paths. We avoid cherry picking convenient stimuli by parametrically generating a set of stimuli to test.

First we enumerate all possible efficient paths an agent could take to reach between one and three of the locations. We repeat this for the version of the map with and without the wall in the middle. Now for each version of the map, create pairs between possible paths going to the same locations, e.g. going to A in the walled version and going to A in the non-walled version. For each pair, create 3 partial path variants. Start from the second to last location they intend to visit, where the agent is 1/3, 1/2, and 2/3 of the way to the final location they intend to visit. If the agent only one has one location they intend to visit, use the starting point as its second to last location.

Now for each pair, keep the partial path variants which change model predictions for either stimulus in the pair. If multiple partial path variants change model predictions in the same way, keep the shortest partial path variant. Using this filter helps to greatly reduce the stimuli set, and makes for a more difficult task—as it biases stimuli toward providing fewer information about paths in order to make an inference. To finalize the set of stimuli, keep pairs for which the top three candidate desires (for the wall/non-wall version) differ in Act.

From those criteria we end up with 8 pairs, and 16 stimuli total. 8 versions of the stimulus with a wall in the middle, and 8 versions of the stimulus without a wall in the middle.

Procedure

These details mirror experiment no. 1 near identically, with the exception that we test on fewer stimuli in this experiment. For good measure, we reiterate the procedure.

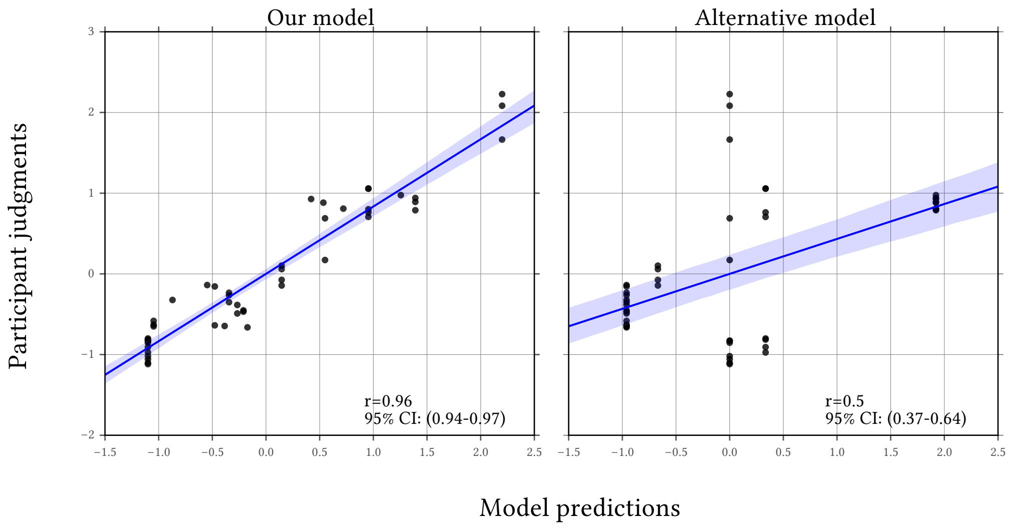


Figure 3-5: Comparison between Act and the alternative model. X-axis are ratings given by models. Y-axis are ratings given by humans. We z-score ratings for comparison.

Participants first read a tutorial which explains the logic of the task. They then complete a survey afterwards to ensure they read the instructions. If they pass, we test them on the task.

During the task, participants complete 16 trials. In each trial participants see a stimulus on the left half of their screen. The stimulus consists of an agent's path toward different objects in a small map, as in Figure 3-1a. On the right half of their screen we ask them to rate their belief for three candidate desires. The ratings scale from 0-10 for each desire. 0 indicates "definitely not", 5 indicates "maybe", and 10 indicates "definitely".

To avoid cherry picking candidate desires which would favor Act, we generate candidate desires with the following procedure. For each trial, normalize the posterior distribution $P(d|a)$ for both Act and the alternative model (which we discuss next). Then for each trial, construct a set containing the top three desires with highest probability for both Act and the alternative model. If on any trial these sets contain duplicate desires, keep the one with the highest probability. Finally, for each trial extract the top three desires. These are the candidate desires we show to participants. With this procedure we avoid choosing desires which strictly favor what Act finds plausible.

Since the candidate desires from Act lack interpretability under appropriate expertise, we translate them into natural language descriptions for our participants. For example, (Or o_1 o_2) would be "the agent wants either o_1 or o_2". To ensure these translations are accurate, two coders translate the natural language descriptions back to their programmatic form in Act—despite being blind to their original form. These coders show full agreement and correctly translate back to the programmatic form in Act for all trials.

Alternative model

Given that this experiment involves an inference from incomplete knowledge of the path an agent takes, we need to revise our alternative model. Our previous alternative model used the following likelihood in its inference scheme.

$$P(a|d) = \begin{cases} 1, & \text{if } a \text{ satisfies } d \\ 0, & \text{otherwise} \end{cases}$$

If an agent is nearly at A but not quite, this likelihood says that the probability that an agent would take those actions if they wanted A is 0. This would not make sensible predictions under incomplete knowledge. Additionally, we saw that this alternative model didn't fit human inferences as strongly as Act. That said, we build a more viable alternative model in the incomplete knowledge context.

$$P(a|d) = \begin{cases} 0, & \text{if } a \text{ satisfies any } i \notin d \\ p, & \text{otherwise} \end{cases}$$

This new alternative model again borrows the syntax of Act, but modifies the semantics of the previous alternative model. It keeps the same prior $P(d)$ but modifies the likelihood $P(a|d)$ to be compatible with incomplete information. First it checks whether the actions satisfy any intentions which aren't intentions of the candidate desire. For example if the agent walks to A and then halfway to B, it's already fulfilled the intention of going to A. Hence the candidate desire of just going to B should be 0. Otherwise if the actions so far are consistent with the intentions of the candidate desire, then $P(a|d)$ is p, where p is the likelihood $P(a|d)$ from Act with a different τ.

Recall that τ is the parameter which controls how an agent factors utility of intentions and actions when making an inference about desires. The closer to 0, the more Act will consider utilities of intentions and actions. But as τ grows, then p reflects the probability of action and intention when these utilities aren't considered. This new alternative model uses a large τ, and as such is agnostic to the utilities of intention and action. Despite not considering utilities, this new alternative model can still aggregate probability for candidate desires under incomplete knowledge. It tracks consistency between actions and the intentions which have been satisfied so far. In this vein, it's an upgrade from the previous alternative model.

3.2.3 Results

Figure 3-5 provides a summary of our results, comparing both Act and the alternative model to human predictions on the desire inference task. Act fits human judgments well, showing a correlation of $r = .96$ with participant judgments (95% CI: $.94 - .97$).

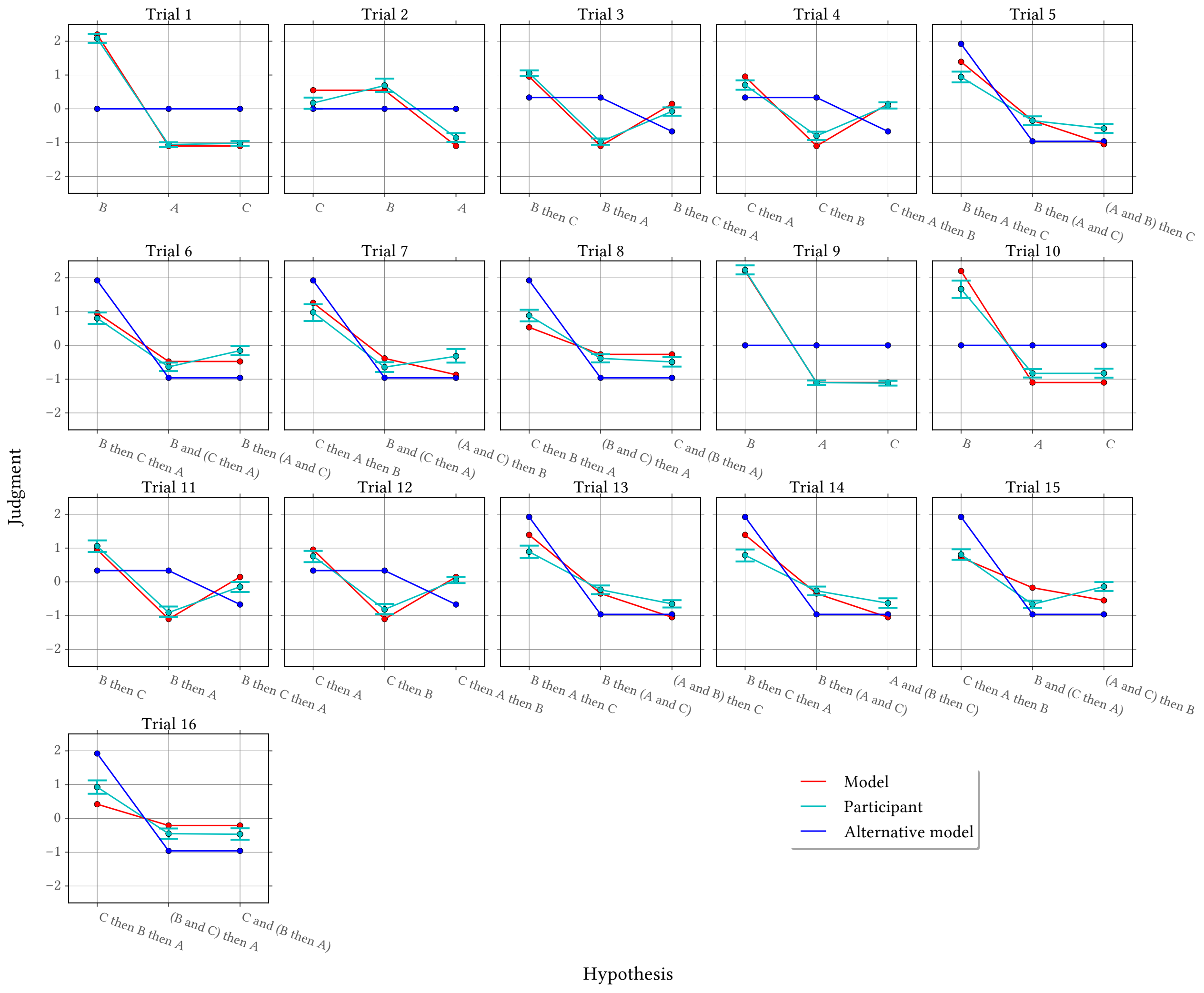


Figure 3-6: Trial by trial comparison between our model, alternative model, and humans on the inference task. X-axis are the candidate desires. Y-axis are the ratings. We z-score ratings for comparison.

For a finer grain look at the results, we show the trial by trial comparison as well in Figure 3-6. For each trial you can see where humans diverge or converge with ACT and the alternative model. Qualitatively, you again see nice convergence between humans and ACT across the trials. On many trials the alternative does not converge as it doesn't fit human predictions well. We take a closer look at why in Figure 3-7, which are the results from the stimulus in Figure 3-1b.

In trial 3, the agent goes to B and then to C. So a sensible response is that they want (THEN B C). Even though there's incomplete information about the agent's behavior, it's clear that they don't want (THEN B A). The steps the agent takes toward C after going to B are suboptimal for a route to A but optimal for C. ACT takes this into consideration when evaluating (THEN B A) as a candidate desire, and rates it highly unlikely in virtue of this. Humans do as well. However, the alternative model does not consider utilities in its inference and so rates (THEN B A) much higher.

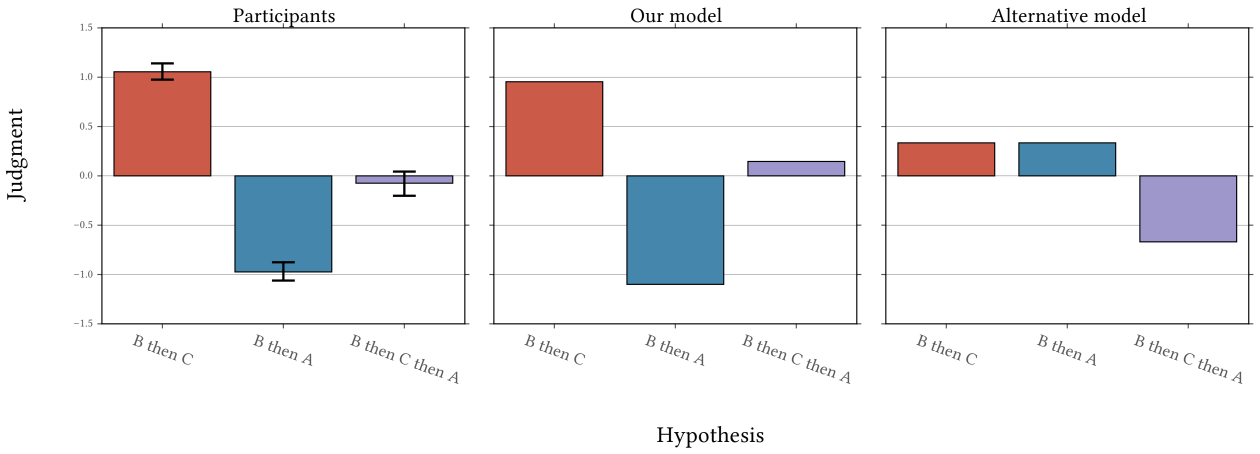


Figure 3-7: In depth look at trial 3. X-axis are candidate desires. Y-axis are the ratings. We z-score ratings for comparison.

3.3 Experiment no. 3

3.3.1 Design

In this experiment we want to know if compositional programs in ACT predict the way humans act on and infer procedural knowledge of the world. The task is similar to the previous experiments but instead focuses on how to infer how some aspect of the world works by observations of someone's behavior. We design an inference task similar to experiment no. 2 where humans observe an agent's partial behavior across one or two days. The distinction is that now the agent's behavior is in service of maintaining a factory. What participants must infer is how to maintain the factory, not what the agent wants. As with the other experiments, they rate how much they think different procedures (desires in Act) correspond to the factory maintenance procedure. From these ratings we recover an approximate distribution to $P(d|a)$ for the desires, intentions, and actions we experimentally test. From it, we can compare to $P(d|a)$ from Act. Recall that $P(d|a)$ uses how humans act on desires in the likelihood $P(a|d)$. As a result, similarities in the empirical $P(d|a)$ and Act's $P(d|a)$ confer predictions about how humans act on and infer desires.

3.3.2 Methods

Participants

30 participants recruited using Amazon's Mechanical Turk Framework.

Stimuli

Figure 3-1c shows an example of the stimuli. They consist of 20 two-dimensional images of an agent who travels to one or more of three potential locations in factory.

We use the same procedure from experiment no. 1 to generate the stimuli. The only exception is that the second map in double events have some starting location and machine locations switched. This is done to fit with the task's cover story, where participants may observe an agent across two different rooms in factory. For good measure, I repeat the procedure.

To construct the single event trials, we enumerate all possible efficient paths an agent could take to reach between one and three of the locations. We then filter this set by removing paths which were equivalent under rotation or reflection of the map.

To construct double event trials, we enumerate all possible efficient paths between one and two of the locations, omitting paths between three locations to prevent the set from growing too large. Unlike the single event trials, we don't remove paths equivalent under rotation or reflection. We require these paths to construct the most primitive desires which occur over two events, e.g. (Or o_1 o_2). This creates a base set of nine paths.

We then split these nine paths into two classes, paths which go to one object and paths which go to two objects. For each class we compute the cartesian product of itself, removing duplicate pairs of stimuli (e.g., the pair $(o_1, o_2) = (o_2, o_1)$). Note that the pair denotes a double event, where each element is an observation on one day. Under a few criteria, we can further pare down the size of the set so it's experimentally tractable.

If pairs violate the principle of rational action, we remove them. For example, when on one day the agent travels to the furthest object from them (so they want that object), and on the second day they grab a different object which is closer to them. Because the map is constant across both days and we assume the agents desire persist across both days, then grabbing the different object on the second day violates rational action.

Other criteria include whether a pair is the reflection or rotation of another pair, e.g. between (o_1, o_1) and (o_3, o_3) we only keep one. And lastly, we remove pairs which only have one plausible desire which explain them. These impact our ability to get graded responses from participants. For example if on both days the agent travels to the furthest object, then it's clear they just want that object. This example won't confer graded judgments from participants, nor from ACT. However, as an exception we include one of these cases just to show that ACT predicts this outcome.

From those criteria there end up being 11 double event stimuli and 9 single event stimuli. In total, these comprise the 20 stimuli we use in the experiment.

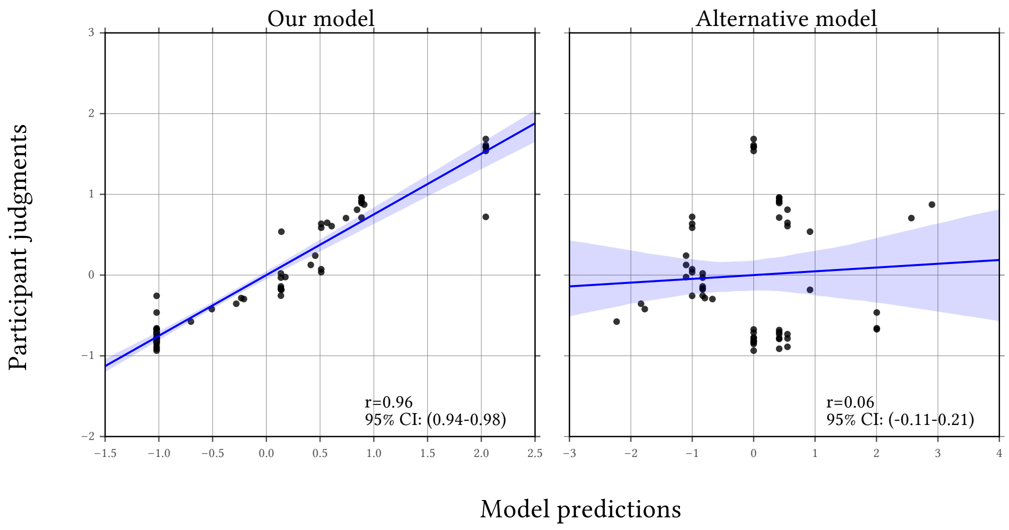


Figure 3-8: Comparison between ACT and the alternative model. X-axis are ratings given by models. Y-axis are ratings given by humans. We z-score ratings for comparison.

Procedure

These details mirror experiment no. 2 near identically, with the exception that our inference task is about procedural knowledge (how to maintain the factory) and not what someones desires.

Participants first read a tutorial which explains the logic of the task. They then complete a survey afterwards to ensure they read the instructions. If they pass, we test them on the task.

During the task, participants complete 20 trials. In each trial participants see a stimulus on the left half of their screen. The stimulus consists of an agent's path toward different machines in a small factory, as in Figure 3-1c. On the right half of their screen we ask them to rate their belief for three candidate maintenance procedures. The ratings scale from 0-10 for each procedure. 0 indicates "definitely not", 5 indicates "maybe", and 10 indicates "definitely".

To avoid cherry picking candidate procedures which would favor Act, we generate candidate procedures with the following procedure. For each trial, normalize the posterior distribution $P(d|a)$ for both Act and the alternative model (which we discuss next). Then for each trial, construct a set containing the top three procedures with highest probability for both Act and the alternative model. If on any trial these sets contain duplicate desires, keep the one with the highest probability. Finally, for each trial extract the top three procedures. These are the candidate desires we show to participants. With this procedure we avoid choosing procedures which strictly favor what Act finds plausible.

Since the candidate procedures from Act lack interpretability under appropriate expertise, we translate them into natural language descriptions for our participants. For example, (Or o_1 o_2) would be "go to either machine o_1 or machine o_2". To ensure these translations are accurate, two coders translate the natural language descriptions back to their programmatic form in Act—despite being blind to their original form. These coders show full agreement and correctly translate back to the programmatic form in Act for all trials.

Alternative model

We reuse the alternative model from experiment no. 2.

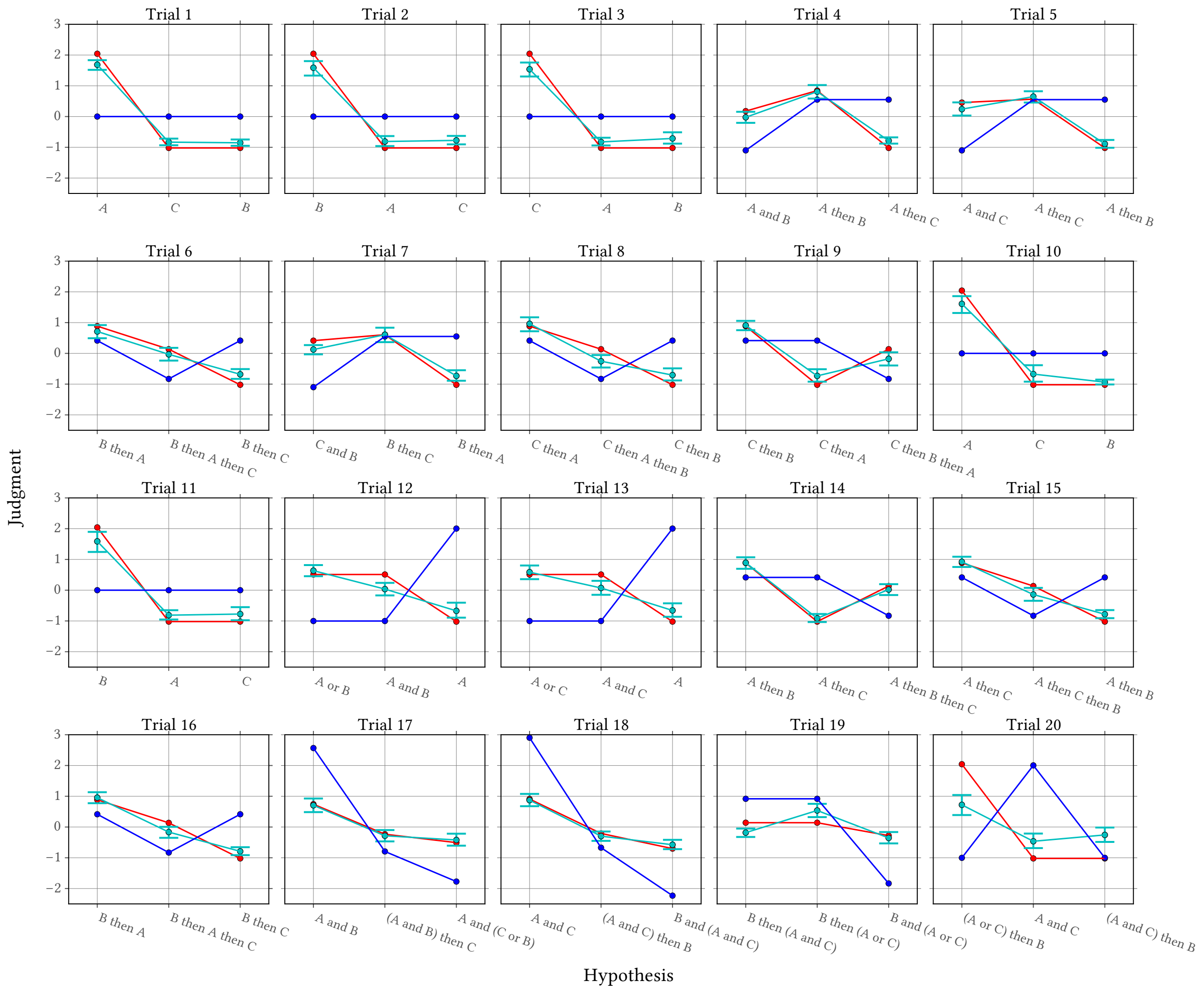


Figure 3-9: Trial by trial comparison between our model, alternative model, and humans on the inference task. X-axis are the candidate desires. Y-axis are the ratings. We z-score ratings for comparison. Legend: red–our model, cyan–participants, blue–alternative model.

3.3.3 Results

Figure 3-8 provides a summary of our results, comparing both Act and the alternative model to human predictions on the desire inference task. Act fits human judgments well, showing a correlation of $r = .96$ with participant judgments (95% CI: $.94 - .98$).

For a finer grain look at the results, we show the trial by trial comparison as well in Figure 3-9. For each trial you can see where humans diverge or converge with Act and the alternative model. Qualitatively, you again see nice convergence between humans and Act across the trials. On many trials the alternative does not converge as it doesn't fit human predictions well. We take a closer look at why in Figure 3-10, which are the results from the stimulus in Figure 3-1c.

In trial 6, the agent goes to B and then to A. So a sensible response is that factory maintenance procedure is (Then B A). Though there's incomplete information, so it's still possible that the procedure could be (Then B (Then A C)). Both Act and participants understand this, but the alternative does not, which relies too heavily on its priors penalizing complexity of a procedure even if it's still plausible. And as we discussed earlier, the alternative model does not integrate utilities into its judgment. Because the agent took a longer path in one room in order to get to B first, it's clear that order is important for the procedure. But without using utilities, the alternative model can't make that inference.

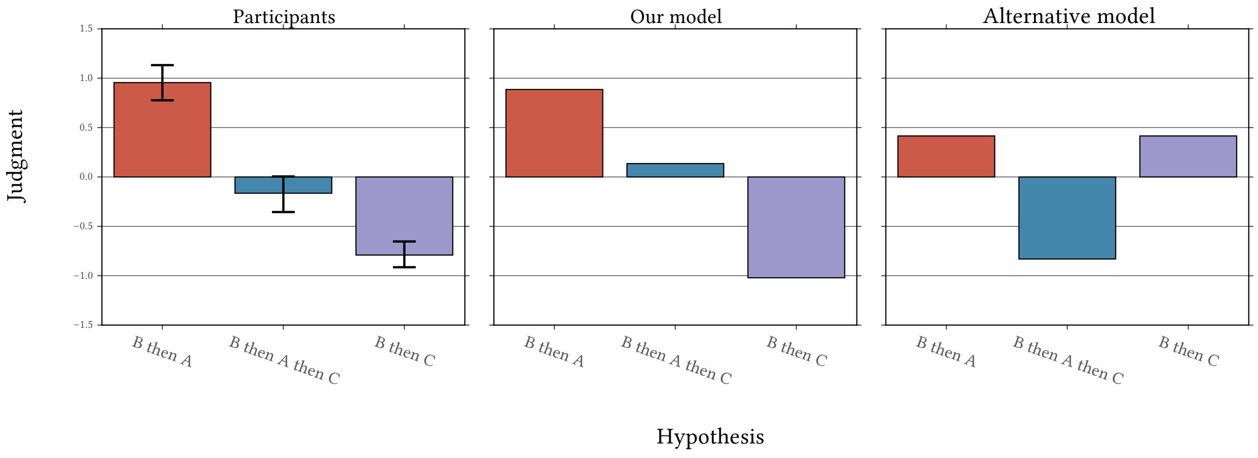


Figure 3-10: In depth look at trial 6. X-axis are candidate desires. Y-axis are the ratings. We z-score ratings for comparison.

Chapter 4

Discussion

4.1 Conclusions and Future Directions

In this work we show that Act predicts how humans act on and infer desires. The strong correspondence between Act and participants across experiments ground this claim.

Our work builds on previous works in grammar-based concept learning [9, 5, 4]. Yet unlike other works, we state our model as a probabilistic programming language—using tools from programming language research. These tools help us be precise about the syntax and semantics of Act. Foremost, these formalities clarify our claim to compositionality. In the denotational semantics we discuss, we show that programs in Act are compositional with respect to intentions and not actions.

Moreover, this work shows how to act on and infer desires which combine constraints on order and choice between objects. This is unlike previous works on how humans act on and infer desires, where the candidate desires do not exhibit such constraints. They are just locations in the world. Despite this progress, we acknowledge that one must go much further. The actions are merely of a gridworld, a simplistic proxy for the environments and decisions that humans face in reality. Extending the current enterprise to more realistic environments and decisions is a fruitful opportunity.

We hope that our approach sets the stage for future efforts, where programming languages take center stage as precise models of human behavior. Throughout the course of this work we design and experimentally test features of the programming language to determine which make it correspond to human behavior. Our alternative models focus on variants in Act's semantics. However we could have also varied Act's syntax. Because our priors in Act are syntax-driven, changing the syntax would change our priors, which would lead to different judgments on the tasks we test. The lesson here is that we can use behavioral experiments to shape programming languages to behave according to human behavior.

4.2 Limitations

I end with some comments on limitations, not just of this work but of similar works in cognitive science.

We try to answer whether humans act like Act. Along the way we glean a number of useful experimental and technical insights on how to treat programming languages as hypotheses for human behavior. However, it's profoundly unclear how to connect these insights to the brain. And it's well within reason to doubt the power of a strictly behavioral approach to studying the brain.

Imagine that we stumbled on our laptops in nature, and in seeking to understand them, we took only a behavioral approach. It's clear that we would be doomed to make sense of our laptops without recourse to the mechanism underlying its behavior. I feel strongly about the same being true of human behavior.

Moreover, while our work paints a particular story about desire and action, there are others. And the ability to disentangle these stories about desire and action rest desperately on the ability to make precise predictions about internal mechanisms that mediate behavior. Because the commitments of many theories today simply don't make strong enough commitments to be falsifiable. If we can think seriously about how to yield neural predictions out of these theories, we could be on our way to progress. Otherwise, I worry we will not.

Bibliography

[1] C. L. Baker, J. Jara-Ettinger, R. Saxe, and J. B. Tenenbaum. Rational quantitative attribution of beliefs, desires and percepts in human mentalizing. *Nature Human Behaviour*, 1(4):0064, 2017.

[2] C. L. Baker, R. Saxe, and J. B. Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

[3] H. P. Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[4] N. D. Goodman, J. B. Tenenbaum, J. Feldman, and T. L. Griffiths. A rational analysis of rule-based concept learning. *Cognitive science*, 32(1):108–154, 2008.

[5] N. D. Goodman, J. B. Tenenbaum, and T. Gerstenberg. Concepts in a probabilistic language of thought. Technical report, Center for Brains, Minds and Machines (CBMM), 2014.

[6] J. Jara-Ettinger, C. Baker, and J. Tenenbaum. Learning what is where from social observations. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 34, 2012.

[7] J. Jara-Ettinger, H. Gweon, L. E. Schulz, and J. B. Tenenbaum. The naïve utility calculus: Computational principles underlying commonsense psychology. *Trends in cognitive sciences*, 20(8):589–604, 2016.

[8] P. D. Mosses. Denotational semantics. In *Formal Models and Semantics*, pages 575–631. Elsevier, 1990.

[9] S. T. Piantadosi, J. B. Tenenbaum, and N. D. Goodman. Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, 123(2):199–217, 2012.

[10] B. C. Pierce and C. Benjamin. *Types and programming languages.* MIT press, 2002.

[11] M. L. Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 2014.

[12] Z. G. Szabó. Compositionality. 2004.

[13] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT press, 1993.