Language Model Agents Enable Semi-Formal Programming

JOSH POLLOCK, ARVIND SATYANARAYAN, and DANIEL JACKSON, Massachusetts Institute of Technology, USA

In the pursuit of malleable programming environments, researchers and practitioners have explored different ways to capture semantic information before a user has fully specified their program. To date, however, these efforts have required system designers to build their formalisms from parts that are either fully formal or else fully opaque to the system. In this paper, we sketch how large language models (LLMs) can enable *semi-formal programming*. In a semi-formal program, some pieces of information are fully formalized in the host language's data structure, but other pieces are left informal. Semi-formal information that is semantically meaningful can be executed directly by a language model agent (LMA), be used to guide the user in fully formalizing their program, or serve as a specification for generating a program that captures the user's intent.

CCS Concepts: • **Human-centered computing** → **Natural language interfaces**; • **Computing methodologies** → *Reasoning about belief and knowledge*; • **Computer systems organization** → **Embedded systems**; *Redundancy*.

Additional Key Words and Phrases: semi-formal programming, live programming, large-language models, language model agents

ACM Reference Format:
Josh Pollock, Arvind Satyanarayan, and Daniel Jackson. 2018. Language Model Agents Enable Semi-Formal Programming. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the pursuit of malleable programming environments, researchers and practitioners have explored different ways to capture semantic information before a user has fully specified their program. For example, the Hazel programming language ensures that even partially specified programs can be evaluated and typechecked [9].

To date, however, these efforts have required system designers to build their formalisms from parts that are either fully formal or else fully opaque to the system. For example, in Hazel partial programs are represented by allowing the programmer to insert a hole. This program hole is either filled with an expression in the formal grammar or else left completely empty, containing no semantic information. Here are two small Hazel programs containing holes:

```
# empty hole: o     # non-empty hole: |"3-5"|
take(o, [1, 2, 3])  take(|"3-5"|, [1, 2, 3])
```

While these programs can technically be executed, they are effectively stuck, and the Hazel runtime cannot make forward progress. Even though the string "3-5" contains semantic information, Hazel is unable to reason about it, because it does not match the formal structure of the surrounding program.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

In this paper, we sketch how large language models (LLMs) could enable a new programming paradigm: *semi-formal programming*. In a semi-formal program, some pieces of information are fully formalized in the host language's data structure, but other pieces are left informal. For example, a user may write `list('3-5', 'Beatles')`, and get the list `['John', 'Paul', 'George', 'Ringo']`. In this example, the function application is formally specified while the arguments (and the function itself) are not. Rather than a specific number or an interval data structure, the user has opted for the less formal `3-5`. The user also passes a string that they intend to be interpreted by the `list` function using background knowledge. Both inputs are specified informally, but contain semantically meaningful information. The `list` function is implemented as a semi-formal program, which allows it to handle these semi-formal inputs.

In pseudocode, a simple implementation of `list` might be the following:

```
def list(number, kind):
    return prompt(f"List {number} {kind}.").parse(List)
```

In contrast to a traditional program where both the input number format and the sorts of things you can list would have to be pre-determined or else rely on hand-crafted logic, a semi-formal program can be defined using a *language model agent* (LMA) (such as GPT) that can reason about the semantics of informal data.

As we discuss in section 2, semi-formal data can be manipulated in three ways. As with the `list` function, semi-formal data can be executed directly by an LMA (a user can **direct** the LMA to execute it); it can be used to guide the user in fully formalizing their program (a user and an LMA can **discuss** the semi-formal information); or it can serve as a specification for generating a program that captures the user's intent (the user can **delegate** a complex, multi-step task to an LMA).

Executing a semi-formal program may require mixing all three strategies. For example if a user is creating a semi-formal web scraper, they may first *delegate* the task to an LMA to generate an initial scraper. Based on this initial result, they may *discuss* with an LMA to surface implicit assumptions they had about their scraping task. Finally, they may use the results of this discussion to *direct* LMAs to execute smaller semi-formal programs such as extracting all paragraphs from a website.

Currently, each strategy is supported by a different environment: LMAs can be *directed* with API calls, *discussions* are carried out with chat interfaces, and *delegation* typically occurs in standalone applications that have access to file systems and external APIs. To support a unified environment for semi-formal programming, we argue system designers should explore the design of *semi-formal representations* that can support each level with the same semi-formal data structures such as structured Markdown files or abstract syntax trees (ASTs) with informal leaf nodes (section 3).

2 AUTHORING AND EXECUTING SEMI-FORMAL PROGRAMS

Because semi-formal programs contain information that cannot be reasoned about by traditional programming languages (PL) methods, they cannot be executed in the traditional sense. Executing a semi-formal program requires a way to reason about the semantics of unformalized sections of that program. Instead of executing these informal sections using an interpreter or translating them using a compiler, we can instead treat them as instructions for an *agent*. We will refer to an agent created using an LLM as a *language model agent* (LMA). In this section, we outline three kinds of interactions users can have with LMAs to operationalize a semi-formal program.

To illustrate these approaches, we use a running example of planning a 36-hour trip to Seattle. Trip planning is a common task that requires users to balance many different considerations (such as the cost of different activities and accommodations, the intensity of their schedule, and competing preferences of group members involved). While these considerations could be written in a formal grammar, preferences typically start out amorphous or even implicit. As a result, eliciting preferences is a significant challenge in addition to resolving conflicts between competing interests.

To develop these levels of interaction, we took inspiration from situational leadership theory, which discusses four levels of interactions between a leader and a follower: directing (leader decides what the follower should do), coaching (leader and follower discuss, then leader decides), supporting (leader and follower discuss, then follower decides), and delegating (follower decides what to do). We decided to center the user in our interaction taxonomy and so opted to let the user lead and the LMA follow. After exploring examples of each level, we decided to merge coaching and supporting into a single level, discussing, to ensure each level is defined by differences in interaction. We thus ultimately settled on three levels of interaction: **direct**—the user interacts with themself, using an LMA as a tool as needed; **discuss**—the user interacts with an LMA to generate, formalize, or execute semi-formal programs; **delegate**—the user defers to an LMA as an autonomous agent, which interacts with itself to execute a complex semi-formal program. These levels are ordered in increasing agency of the LMAs involved. LMAs are **directed** when the task has little ambiguity, and LMAs are **delegated** to when the task requires a model to make many different decisions to resolve ambiguities or procure information.

2.1 Direct

At one end of the agency spectrum, a user can *direct* an agent to execute semi-formal instructions. For example, after gathering a list of restaurants from Eater.com,[1] a user might write the following semi-formal program:

```
pick('a great restaurant for vegetarians near Seattle', [
    """## Ono Authentic Hawaiian Poke
       Ono, named for Oahu-born owner Steven Ono, is a seafood lover’s dream in Edmonds...""",
    """## Cafe Juanita
       Kirkland’s Northern Italian fine dining mainstay offers several excellent seasonal...""",
    ...])
# AnswerDirectly('Plum Bistro')
```

The pick function could be implemented using a semi-structured prompt as follows:

```
def pick(criteria, items):
    prompt(f"""You are PickOneBot. Pick one of the items in the input list. It should be {criteria}.
# Response Format
Your response should have two parts: a Thought and an Action.
First think about what to do, and respond in this format:
## Thought
(Briefly describe your thought process.
Let's work this out step by step to be sure we take the right action.)

Then respond with one of the following action formats:
## AnswerDirectly (Choose this option to provide your answer directly.)
Or
## CannotAnswer (Choose this option if you cannot answer the question. Please explain why.)

Begin!
# Input
{input}""").parse()
```

[1]Restaurant descriptions from https://seattle.eater.com/maps/best-restaurants-seattle-38.

The call to `pick` is a semi-formal program, since the function application is formally defined in the host language, but its inputs encode semantic data that cannot be reasoned about by the host language. The implementation of `pick` is also a semi-formal program, containing semi-formal information within a structured Markdown prompt.

Semi-formal functions like `pick` can be combined by end-users in programming or low-code environments to construct larger programs. In these kinds of interactions, the user is primarily in a loop with themselves (similar to traditional live programming). They write mostly formal code, and use the LMA as a tool to execute small, precise, semi-formal programs that typically require only one forward pass of an LLM. As a result, these programs can typically be executed quickly and the LMAs will tend to produce high quality output. On the other hand, using an LMA in this way places more burden on a user to specify the semi-formal program themselves. For instance, at the cost of providing a list of restaurants from Eater explicitly, the end user avoids potential restaurant hallucinations or outdated information and simplifies the task from generating a restaurant to selecting one.

2.2 Discuss

Instead of working on their own to write semi-formal programs, a user can also work cooperatively with an LMA to gradually specify a semi-formal program. For example, a user might ask ChatGPT to help them plan a trip. Discussions can be further broken down into *consulting* and *supporting* interactions (inspired by situational leadership theory).

Here's an example ChatGPT response when a user asks for help planning a trip:

> Sure, I'd be happy to help you plan your trip! However, as of my knowledge cutoff in September 2021 and due to my current constraints, I'm unable to access real-time data or internet to provide you with the current weather forecast. I recommend checking a reliable weather website or app for the most accurate and up-to-date forecast.
>
> For the purpose of trip planning, I can still provide recommendations based on typical weather patterns in Seattle during different times of the year. To do this, could you please tell me the month and general dates of your trip?
>
> (1) What are your main interests? (e.g., art, history, nature, technology, music, food, sports)
> (2) Are there any specific places in Seattle you already plan to visit?
> (3) ...

Here the user *consults* with the agent about how to plan their trip. Though ChatGPT later provided a suggested itinerary after a short conversation, its suggestion was generic and unable to take advantage of up-to-date information about new restaurants, relevant weather, or upcoming events. The value added by the LMA in this scenario was not the final semi-formal trip itinerary, but rather helping the user elicit implicit knowledge such as specific kinds of activities they wanted to do.

In the scenario above, the user is still ultimately responsible for planning their trip. They use the LMA as a guide to augment their decision-making process. If the agent is instead elevated to be the primary planner of the trip, the user shifts from *consulting* the LMA to merely *supporting* it in generating an itinerary. For example, by augmenting ChatGPT with plugins like OpenTable, ChatGPT can take a series autonomous actions before arriving at an answer. This shifts some of the planning responsibility towards the LMA. However, the user is still able to significantly guide its direction. By placing more responsibility on the LMA, the user's responsibilities shift from writing semi-formal programs to merely reading them. As a result, *supporting* interactions may be more accessible than *consulting* to end-users without extensive domain knowledge or programming expertise. On the other hand, shifting more agency to the LMA can result in more hallucinations.

2.3 Delegate

At the extreme autonomous end of the spectrum, a user can choose to delegate an entire complex task to an agent. One such system that tries to do this is Auto-GPT [1]. With Auto-GPT, a user provides a prompt for a complex task, like "plan a trip to Seattle," and Auto-GPT creates a plan and executes it autonomously with dozens of LLM calls and external actions such as reading and writing to files and searching the internet.

Here is a subset of the goals Auto-GPT generated when asked to plan a Seattle trip in the style of the New York Times' 36 hour articles:

(1) Create a detailed and engaging 36-hour itinerary for Seattle, including a variety of attractions, local landmarks, and unique experiences, in line with the style of the New York Times' 36 Hours series.
(2) Provide a curated list of dining options, ranging from local favorites to high-end restaurants, ensuring a diverse culinary experience throughout the trip.
(3) Ensure the itinerary is feasible and efficient, taking into account travel times, opening hours, and other logistical considerations.

This semi-formal program was authored and then executed completely autonomously. While this approach has some promise and can successfully produce a travel itinerary, fully autonomous LMAs are not suitable for large, ambiguous problems [10]. This is because such problems require more user interaction to clarify intent. Without such prompting, systems like Auto-GPT tend to output generic solutions. Long-running self-agent executions also risk diverging into infinite loops or racking up high costs. The output of a delegated task can also be harder to modify, since such tasks typically produce larger results (a whole app, for example) that require more time, effort, and expertise to review.

2.4 Moving Between Levels By Centering Semi-Formal Representations

While we introduced the levels independently, we believe that users are likely to want to switch between different levels throughout a single task. For example, a user might first want to **delegate** the task of creating a basic itinerary to an agent that can search the internet. But afterwards, they might want to customize the result. Maybe they want an agent to pick a restaurant from Eater. Or maybe they realized they preferred hiking to museums after reading the initial itinerary. These tasks are closer to **directing** and **discussing** with the LMA. Alternatively, a user might want to start by **discussing** their travel ideas with an LMA to refine their intentions before **delegating** the task to the LMA to complete.

Currently, LMA interfaces are designed around just a single mode of interaction. **Directing** is most easily accomplished with API calls. The ChatGPT interface is convenient for **discussing**, and Auto-GPT is useful for **delegation**. This siloing makes it difficult to switch between modalities. We propose instead to center LMA interfaces around shared *semi-formal representations*. A semi-formal representation of a trip, for example, might be a list of activities for each day with rough time windows, locations, and weather projections. By holding the representation stable we thus allow the interaction mode to shift over the course of a task. With a single representation, a user could generate an initial itinerary by **delegating** the task to an LMA, **direct** an LMA to replace a specific location, and **discuss** with an LMA that could offer suggestions for changes they might want to make.

The idea of centering a representation is not new. As Heer discusses in "Agency plus automation: Designing artificial intelligence into interactive systems" [5], a domain-specific language (DSL) such as Vega-Lite can serve as a shared representation both end-users and agents. However, while previously agents and direct manipulation environments had to be painstakingly designed for each new DSL, LMAs introduce the possibility of dynamically generating task-specific views of a DSL as needed either automatically or with the help of prompt engineering [7].

3 TOWARDS FORMALIZING SEMI-FORMAL PROGRAMMING

```
type SemiFormal =
| Informal(string)
| Primitive(string | number | boolean)
| Union(Type[])
| Product(Type[])
| Function(Type, Type)
```

Fig. 1. A type for semi-formal programs.

```
type Pick = (criteria: Informal, items: Informal[]) =>
(AnswerDirectly(Informal) | CannotAnswer(Informal))
```

Fig. 2. The type of the pick function in subsection 2.1.

What does a semi-formal representation look like? We have already seen some examples in section 2. In our **direction** example, we used *structured Markdown* as a semi-formal representation of the input data. Markdown affords hierarchical nesting using headers while also affording informal text regions. Additionally, Markdown allows one to encode formalized programs in code fences. These affordances may explain why Markdown has become a popular format for interacting with LMAs. LMAs are also able to use semi-formal representations for their output. For example, in subsection 2.2 ChatGPT responded with a bulleted list of questions. This structure both improves readability and, when an LMA is explicitly guided to produce structured output, allows a traditional program to parse the structure latent in informal pieces of semi-formal programs.

Figure 1 offers a more precise semantic definition of a semi-formal program. By augmenting a normal program with an `Informal` primitive, we can type functions such as the `pick` function from subsection 2.1. We provide this type in Figure 2. `Informal` plays a similar role to `any` in gradual typing systems like TypeScript. However, unlike `any`, `Informal` implies that the data contains meaningful semantic content that can be interpreted by an LMA.

While our `SemiFormal` type gives some shape to semi-formal programming, it is only the beginning of a comprehensive theory. A big open question remains: How can we find structure in the informal parts of semi-formal programs? The more structure we can get, the more we may be able to transfer traditional PL approaches to this new style.

One promising direction is probabilistic programming languages. Wong et al. used LMAs to translate natural language statements into probabilistic programs that can explicitly represent ambiguity present in the source material [11]. This allows them to apply traditional symbolic probabilistic programming techniques to ambiguous information.

Another potential source of inspiration is research on structured natural languages. For example, although they are often derided, *jargon* like legalese, medicalese, and academese are forms of domain-specific natural languages [6, 8]. They have rich structure, but are not necessarily formal in the PL sense. Their utility largely comes from their ability to communicate information quickly and precisely. Is it possible to find jargon for different domains that can be reliably translated to DSLs with the help of LMAs?

The natural semantic metalanguage (NSM) offers a third potential source of formalism [2, 3]. The NSM comprises a set of culturally universal primitive words like "someone," "good," and "place," that can be used to define complex culturally specific words like "love" and "friend." In addition, the NSM can be used to define *cultural scripts*, which express culturally specific values [4]. The NSM suggests a path to defining semi-formal intermediate representations based on unambiguous primitive terms that could be translated across languages, edited, or recombined.

REFERENCES

[1] Toran Bruce Richards. 2023. Auto-GPT. https://github.com/Significant-Gravitas/Auto-GPT.
[2] Cliff Goddard. 2009. The natural semantic metalanguage approach. (2009).

[3] Cliff Goddard. 2018. *Ten Lectures on Natural Semantic Metalanguage: Exploring language, thought and culture using simple, translatable words*. Vol. 21. Brill.
[4] Cliff Goddard and Anna Wierzbicka. 2004. Cultural scripts: What are they and what are they good for?
[5] Jeffrey Heer. 2019. Agency plus automation: Designing artificial intelligence into interactive systems. *Proceedings of the National Academy of Sciences* 116, 6 (2019), 1844–1850.
[6] Russel Hirst. 2003. Scientific jargon, good and bad. *Journal of technical writing and communication* 33, 3 (2003), 201–229.
[7] Geoffrey Litt. 2023. Malleable software in the age of LLMs. https://www.geoffreylitt.com/2023/03/25/llm-end-user-programming.html.
[8] LC Mugglestone. 1995. Jargon: Its Uses and Abuses. *Notes and Queries* 42, 1 (1995), 87–89.
[9] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.
[10] Shawn "swyx" Wang. 2023. *LangChain "Chains vs Agents" Webinar - it's a smol world*. YouTube. https://www.youtube.com/watch?v=bYLHklxEd_k
[11] Lionel Wong, Gabriel Grand, Alexander K. Lew, Noah D. Goodman, Vikash K. Mansinghka, Jacob Andreas, and Joshua B. Tenenbaum. 2023. From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought. arXiv:2306.12672 [cs.CL]