QUASAR: A Programming Language Specialized for LLM Code Actions

Stephen Mell[1,2], Botong Zhang[2], David Mell[3], Shuo Li[2], Ramya Ramalingam[2], Nathan Yu[3], Stephan Zdancewic[2], Osbert Bastani[2]

[1]Carnegie Mellon University
stephenmell@cmu.edu

[2]University of Pennsylvania
{bzhang16,lishuo1,ramya23,stevez,obastani}@engineering.upenn.edu

[3]Independent Researcher
zraexy@gmail.com, nathany@alumni.cmu.edu

Abstract

Large language models (LLMs) often call external tools to solve tasks. One effective strategy is for LLMs to write code, enabling them to use complex control flow such as conditionals and loops. Such *code actions* are typically represented as Python code, since LLMs are proficient at writing it. However, many programming language features that would support more effective code actions are difficult to implement for Python. We propose separating *internal code* that captures program logic from *external calls* to tools that interact with the world. New features can then easily be implemented by (1) annotating external calls with the effects relevant to that feature, and (2) modifying the execution of the internal code to track this information. We develop a novel programming language, QUASAR, that implements this idea. To illustrate its utility, we implement several useful features on top of QUASAR to enhance code actions: access control with batched user queries to improve security, autoparallelization of external calls to reduce latency, and conformal prediction for uncertainty quantification to mitigate hallucinations.

1 Introduction

Large language models (LLMs) have demonstrated remarkable reasoning capabilities. A powerful way to augment these capabilities is via *tool calls* (Schick et al., 2023), which let the LLM write function calls whose results are appended to the context before generation continues. Tools include functions to read and edit files (Yang et al., 2024), query knowledge sources such as databases (Lewis et al., 2020), store information in external memory across interactions (Liu et al., 2024b; Maharana et al., 2024), and access user input/output devices such as the mouse, keyboard, and screen (Anthropic, 2024). Modern approaches often combine tool calling with chain-of-thought reasoning (Wei et al., 2022) to create autonomous *agents* capable of solving problems through sequential tool calls (Yao et al., 2023b).

Recently, there has been a shift toward providing tools in the form of software APIs and having LLMs write code that invokes them, an approach known as *code actions* or *programmatic tool calling*. This strategy enables the LLM to write code that includes control flow to facilitate more complex interactions, such as automating iterative tasks by writing loops, without the result of every tool call needing to pass through the LLM context. Code actions have been shown to provide better task performance, have lower latency, and use fewer tokens than traditional tool calling (Wang et al., 2024; Wu, 2025; Zhang et al., 2025a), and even to provide protection against prompt-injection attacks (Debenedetti et al., 2026).

Existing systems predominantly use Python (Wang et al., 2024) and Bash (Anthropic, 2026; Yang et al., 2024) for code actions, since the prevalence of these languages makes it easy

Is there an alcoholic drink in this image?

(a)

```
drink_patches = image_patch.find("drink")
found = False
for drink_patch in drink_patches:
    if drink_patch.simple_query(
        "Does this have alcohol?"
    ) == "yes":
        found = True
return found
```

(b)

Figure 1: Illustrative example (Surís et al., 2023) of an image and a natural language question about that image (a), and a code action answering that question (b).

for LLMs to write code in them (Cassano et al., 2023; Orlanski et al., 2023). However, these kinds of general-purpose languages were not designed with code actions in mind and lack potentially useful features—for instance, access control restrictions (for calls to irreversible APIs that might require human approval) to improve security, autoparallelization to improve latency, and uncertainty quantification (for code actions calling other models, e.g., sub-agents) to improve reliability.

Furthermore, implementing these features directly in general-purpose languages is complicated by their design. Specifically, we observe that code actions combine two distinct kinds of computation: *internal code*, which performs complex logic but has no effect on the world (e.g., data manipulation or control flow), and *external calls*, which affect the world (e.g., sending emails or invoking other models). Consider the example in Figure 1, where the program invokes sub-agents (`find` and `simple_query`) to answer the question. The **`for`** loop, **`if`** statement, and computation of `found` are the internal computations, which orchestrate the external calls to sub-agents.

General-purpose languages such as Python and Bash do not separate external calls from internal code. Yet, doing so can make it significantly easier to implement desirable language features. At a high level, annotations document the relevant effects of external calls (e.g., resources accesses, data dependencies, or prediction uncertainties), and internal code tracks these effects during execution. In our example, suppose we want to apply conformal prediction (Vovk et al., 2005) to quantify uncertainty. The external calls to sub-agents are modified to return sets of labels, and the execution of the internal code is modified to operate on these sets (e.g., Booleans are now **`True`**, **`False`**, or {**`True`**, **`False`**}). This approach can also be applied to access control, where a key challenge is approval fatigue (Beautement et al., 2008; Hughes, 2026). Traditional languages block execution until actions are approved by the user, forcing sequential approvals. The number of user interactions can be reduced by presenting actions for approval in large batches. To realize this approach, external calls can be annotated with the resources they access, and internal code can be "opportunistically" executed (Mell et al., 2025) to batch as many external calls as possible. Users are only asked for approval when no further progress can be made without it. Opportunistic execution also enables autoparallelization, where external calls are labeled with data dependencies and internal code executes external calls in parallel when possible. To some degree, these features can be implemented in existing languages via "tracing", executing code to build computation graphs (Reed et al., 2022). However, tracing-based approaches rely on the host language for control flow and thus do not capture it. This limits the kinds of features that can be supported, e.g., preventing conformal prediction from executing both branches of an **`if`** statement when the condition is {**`True`**, **`False`**}.

We propose a new language called QUASAR[1] that explicitly separates internal code from external calls and serves as a platform where features supporting code actions are simple to implement. QUASAR can support arbitrary Python libraries by treating them as external

[1]QUASAR: QUick And Secure And Reliable

tool calls. To enable LLMs to generate QUASAR code, we rely on recent work compiling an expressive subset of Python into λ calculus (Mell et al., 2026a), and restrict the LLM to generating code within this subset. To demonstrate the utility of QUASAR, we implement and evaluate three diverse features: batched resource approval for improved access control; autoparallelization of external calls; and conformal semantics for uncertainty quantification.

In summary, we make the following contributions:

- We propose distinguishing between internal computation and external tool calls in code actions, and we design a language specialized for this.
- We implement this idea as QUASAR, a flexible platform for alternative program semantics and effect annotations.
- We show its extensibility by implementing and evaluating three diverse features: access control approval batching, autoparallelization, and conformal prediction.
- We show how to generate QUASAR code from LLMs without reducing performance on downstream tasks.

Related work. Numerous studies have explored augmenting LLMs with code actions to improve task performance, including CodeAct (Wang et al., 2024), ViperGPT (Surís et al., 2023), and AppWorld (Trivedi et al., 2024). Despite these advancements, several challenges remain, including security and privacy risks (He et al., 2026; Andriushchenko et al., 2025), persistent hallucination issues (Liu et al., 2024a; Li et al., 2024a), and concerns over computational efficiency (Yao et al., 2023a). Recent work has proposed leveraging code actions to improve security, where LLM-generated code can be analyzed for security vulnerabilities (Debenedetti et al., 2026). However, it does not support batching of user approval requests, and it is not extensible (e.g., for latency or reliability improvements). Other work has addressed conformal prediction of functional programs with neural components (Ramalingam et al., 2024) and automatic parallelization of functional programs (Mell et al., 2025). Though we draw on insights from this work, neither considers programs generated by LLMs nor imperative features of languages such as Python. Finally, QUASAR relies on the compiler proposed in Mell et al. (2026a); that work focuses exclusively on autoparallelization, does not provide the broader benefits of QUASAR for implementing code actions, and does not show how to generate programs in its supported subset of Python.

2 QUASAR programming language

We first describe the syntax and semantics of QUASAR programs (§2.1); then, we describe how QUASAR code is generated by LLMs (§2.2). We show a running example in Figure 2 for the problem in Figure 1.

2.1 Syntax and semantics

A QUASAR program $P \in \mathcal{P}$ consists of standard syntactic constructs such as conditionals, loops, and function calls. The execution of a QUASAR program $P \in \mathcal{P}$ is expressed as a set of *rewrite rules* $\mathcal{R}$. If a rule $R \in \mathcal{R}$ is applicable to P, then it transforms P into a new program P', which we denote by $P \rightarrow_R P'$. In general, there may be multiple possible programs P' satisfying $P \rightarrow_R P'$, such as if the rule R is applicable to different parts of P. If there is any rewrite rule R mapping P to P', then we simply write $P \rightarrow P'$. We give the full set of rewrite rules in Figure 8 in Appendix A. The core semantics does not require that these rules be applied in a particular order; regardless of the order, the program result will always be the same (a property known as *confluence*). Thus, QUASAR extensions are free to choose whatever order is most appropriate.

There are two kinds of rewrite rules: internal rules $\mathcal{R}_{\text{int}}$ and external rules $\mathcal{R}_{\text{ext}}$. Internal rules do not have *effects*, meaning they do not have consequences external to the program, including network calls, system calls, calls to external APIs, or even printing. Internal rules perform transformations such as substituting variables, unrolling loops, and resolving

$P_1 =$
```
drink_patches = image_patch.find("drink")
found = False
for drink_patch in drink_patches:
    if drink_patch.simple_query(
        "Does this have alcohol?"
    ) == "yes":
        found = True
return found
```

$E_1 = \{(\texttt{image_patch.find("drink")}, \varnothing)\} \quad \rightsquigarrow$
$E_2 = \{(\texttt{image_patch.find("drink")}, \texttt{[patch1, patch2]})\}$

$P_2 =$
```
found = False
for drink_patch in [patch1, patch2]:
    if drink_patch.simple_query(
        "Does this have alcohol?"
    ) == "yes":
        found = True
return found
```

$P_3 =$
```
found = False
if patch1.simple_query(
        "Does this have alcohol?") == "yes":
    found = True
if patch2.simple_query(
        "Does this have alcohol?") == "yes":
    found = True
return found
```

$E_3 = \{(\texttt{patch1.simple_query(...)}, \varnothing),$
$(\texttt{patch2.simple_query(...)}, \varnothing)\} \quad \rightsquigarrow$
$E_4 = \{(\texttt{patch1.simple_query(...)}, \texttt{"no"}),$
$(\texttt{patch2.simple_query(...)}, \texttt{"yes"})\}$

$P_4 =$
```
found = False
if "no" == "yes":
    found = True
if "yes" == "yes":
    found = True
return found
```

$P_5 =$ `return True`

Figure 2: Given program P_1 from Figure 1, one way QUASAR could execute it is as follows. First, it dispatches the external call `image_patch.find("drink")`, resulting in execution set E_1. Eventually, this external call finishes running and returns `[patch1, patch2]`, resulting in execution set E_2, after which QUASAR applies R_{ext} to substitute this value into P_1 to obtain P_2. Then, QUASAR applies an internal rule to unroll the for loop in P_2 to obtain P_3. Next, it dispatches both `patch1.simple_query(...)` and `patch2.simple_query(...)` resulting in execution set E_3. As before, these external calls finish running and return `"no"` and `"yes"`, respectively, yielding E_4, so QUASAR applies R_{ext} twice (once for each external call) to substitute these values into P_3 to obtain P_4. Finally, QUASAR applies additional internal rules to simplify the conditionals in P_4, resulting in terminal program P_5.

conditionals; these rules are applicable when the necessary values are constants (e.g., a conditional whose predicate is **`True`** or **`False`**).

For example, in program P_2 in Figure 2, the list in the for loop is a constant value `[patch1, patch2]`, so QUASAR applies a rule to unroll the for loop, resulting in P_3. Similarly, in P_4, it can apply rewrite rules to transform the predicate `"no" == "yes"` into **`False`** and the predicate `"yes" == "yes"` into **`True`**, after which it can rewrite the conditionals to obtain P_5.

There is only one external rule $\mathcal{R}_{\text{ext}} = \{R_{\text{ext}}\}$. This rule is designed to enable calls to *external functions* $f \in \mathcal{F}_{\text{ext}}$. Unlike a typical function, which is implemented as QUASAR code, an external function is implemented in Python; thus, external functions can perform desirable effects such as printing a value or calling an LLM to obtain its output. An *external call* in program P is a statement $S = y \leftarrow f(x_1, ..., x_k)$ that calls an external function $f \in \mathcal{F}_{\text{ext}}$.

QUASAR may execute external calls as soon as all their arguments are available. Formally, an external call $S = y \leftarrow f(x_1, ..., x_k)$ in a program P is *dispatchable* if all of $x_1, ..., x_k$ are values (e.g., `0`, **`True`**, or `"foo"`; recall that variables become values as the program is incrementally rewritten). As QUASAR performs rewrites, it keeps track of the currently executing external calls $(S, B) \in E$, where S is a pointer to the external call in the current program P (preserved by rewrites) and B is a pointer to a value that is initially $\varnothing$ but is eventually set to the output of the external function. After a rewrite $P \rightarrow P'$, QUASAR identifies all the dispatchable external calls S in P' that are not yet in E; for each $S = y \leftarrow f(x_1, ..., x_k)$, it executes f on $x_1, ..., x_k$ in a separate thread T, and adds the pending call (S, B) to E. The thread T is also given B; once it finishes executing the external function f, it writes the output of f to B and terminates. Then, QUASAR applies the rewrite rule R_{ext} to the current program (which may no longer be P') to substitute the value in B into the program.

```
({77: '.find', 83: '.simple_query'},
 ('def',
  75,
  ((76,),
   ((('prim', 78, 'drink'),
     ('call', (79,), 77, (76, 78)),
     ...
         ('call', (91,), 0, (85,)),
         ('call', (92,), 91, (86, 88)),
         ('call', (90,), 92, ())),
        (90,)))),
     ('call', (93,), 79, (80, 81))),
    (93,)))))
```

(a)

Approach	Execution		Accuracy	
	GQA	AD	GQA	AD
Python	99.6	76.3	71.8	64.5
Ours	97.9	82.8	71.4	63.4
Multi-turn	99.5	89.2	72.3	67.7
Nano Base	92	–	65	–
Nano SFT	99	–	71	–

(b)

Figure 3: The Python code from Figure 1 as QUASAR code (a). Comparison of different code generation approaches on the GQA and AgentDojo (AD) datasets (b). "Execution" is the fraction of generated programs that execute successfully (i.e., no syntax or runtime errors). "Accuracy" is the fraction of generated programs that output the ground truth label.

For example, in Figure 2, given the initial program P_1, QUASAR executing with the parallel semantics immediately dispatches the external call `image_patch.find("drink")` in a separate thread, leading to execution set E_1. When this thread finishes, it writes the result `[patch1, patch2]` to B, resulting in E_2. This allows R_{ext} to be applied to P_1, obtaining P_2. Similarly, as soon as QUASAR rewrites P_2 to P_3, it dispatches two external calls `patch1.simple_query(...)` and `patch2.simple_query(...)`, resulting in E_3; these execute and return `"no"` and `"yes"`, respectively, resulting in E_4. Finally, QUASAR applies R_{ext} twice to substitute these values into P_3, resulting in P_4. A program P is *terminal* (i.e., finished executing) if no rules are applicable to P and there are no pending external calls. P_4 is then rewritten to P_5, which is terminal.

2.2 Generating QUASAR code

Though, for illustration, we have written example code with a syntax similar to Python, raw QUASAR code actually looks very different. A key challenge is that LLMs have never seen it before, and we find that they struggle to generate it directly. Instead, our strategy is to have the LLM generate Python and then *transpile* this Python code to QUASAR using an existing compiler (Mell et al., 2026a). That is, the LLM generates the code in Figure 1, we transpile it to the code in Figure 3a, and then the QUASAR interpreter executes it. It is very challenging to transpile unrestricted Python to QUASAR, since doing so would inherit all the challenges of applying alternative semantics to Python. However, many practical code actions do not use the unsupported language features of Python (e.g., classes and inheritance); intuitively, LLMs are trying to perform actions, not write complex software. Thus, our transpiler supports a restricted subset of Python carefully chosen to balance expressiveness and ease of transpilation; we provide details on this transpilation in Appendix B. We consider three strategies for generating Python code in this subset. (1) *Instruction:* It is often sufficient to instruct the LLM to do so in the system prompt. (2) *Multi-turn:* It is also possible to feed error messages from our transpiler back to the LLM in a multi-turn loop. Importantly, this feedback is based only on static program information; in contrast, traditional multi-turn feedback based on Python interpreter errors requires running the code, which can result in undesirable side effects from unsuccessfully executed code. However, a shortcoming of this approach is that calling the LLM multiple times can be slow. (3) *Supervised fine-tuning:* For smaller models that struggle to adhere to the allowed subset of Python, we can use supervised fine-tuning to improve adherence.

We show that generating QUASAR code via transpilation retains task performance comparable to that of Python, and we present several strategies for improving transpilation success. Specifically, we compare ordinary, unrestricted Python generation ("Python") to Python generation restricted to the allowed subset, via simple prompt instructions ("Ours"), as well as using multi-turn feedback ("Multi-turn"). Finally, we consider fine-tuning a small model (i.e., GPT-4.1-nano) using programs generated by a large model (i.e., GPT-5). We report results for both the base small model ("Nano Base") and the fine-tuned version ("Nano

Algorithm 1: Pseudocode for the batched execution semantics.

```
function RunBatchedApproval(P)
    while P has dispatchable calls do
        Identify dispatchable external calls {S} in P
        Query user to validate {S}
        Terminate execution if rejected
        Dispatch {S} and add to a set E
        P ← RunInternal(P, E)
    return P

function RunInternal(P, E)
    while true do
        if exists (y ← f(x_1, ..., x_k), B) ∈ E and B ≠ ∅ then
            apply R_ext to P to substitute B in for y
        else if exists rule R ∈ R_int applicable to P then
            apply R to P
        else if E = ∅ then
            break
    return P
```

SFT"); for these results, we use 900 examples for training and report results on a held-out subset of 100 examples. Figure 3b shows, for each approach and dataset, the fraction of tasks where the generated program executes without error ("Execution") and where it produces the correct result ("Accuracy"). Execution errors can be due to the LLM failing to adhere to the allowed subset or due to runtime errors. Restricting to the Python subset ("Ours") does not significantly change accuracy for either dataset, while multi-turn feedback actually improves accuracy for both datasets ("Multi-turn"). Finally, SFT improves accuracy for GQA; the AgentDojo dataset contains only 93 tasks, which is not enough for SFT.

3 Enhancing code actions by extending QUASAR

To demonstrate how features supporting code actions can easily be implemented in QUASAR, we implement three such features: access control with batched requests, autoparallelized execution of external calls, and conformal prediction for uncertainty quantification.

3.1 Batching access control queries

We consider a standard security model based on access control (Sandhu & Samarati, 1994; Sandhu, 1998), where the user must approve the execution of effects. Because effects are isolated in external calls, we only need to ensure that external calls are consistent with the user's desired security policy. For instance, a smartphone user might grant an app access to resources such as their location and the ability to send emails, in which case the app would be allowed to access only these resources.

Because of the large space of actions and the uncertain risk tolerance of users, many systems, such as the coding agent Claude Code (Anthropic, 2026), dynamically ask users to approve each external call. A key challenge with dynamic access control is that queries can block progress while waiting for the user to approve requests, preventing agents from working autonomously. Thus, we implement a *batched-approval semantics* that collects as many external calls as possible and queries the user to confirm them simultaneously (Algorithm 1). If the user rejects them, execution terminates; otherwise, the external calls are dispatched and execution proceeds. RUNINTERNAL performs as many rewrites of the current program P as possible (including both applying internal rules and handling previously dispatched external calls). It returns once P cannot be rewritten any further, at which point the main routine RUNBATCHEDAPPROVAL queries the user to validate all the dispatchable external calls in P, and then dispatches all of these calls. This loop continues until P is terminal. For example, in Figure 2, QUASAR would ask the user for permission to make the external call `image_patch.find("drink")` in P_1, but would then be able to batch the permission requests for `patch1.simple_query(...)` and `patch2.simple_query(...)` in P_3.

3.2 Autoparallelizing external calls

Many external tool calls (e.g., recursive calls to sub-agents) execute very slowly; thus, to reduce latency, they would ideally be executed in parallel where possible. The design of QUASAR is naturally suited to this: because programs are interpreted using rewrite rules, a statement can be "executed" as soon as the relevant program variables are substituted with constants, allowing statements to run out of order and in parallel. The example in Figure 2 is actually executing with QUASAR's *autoparallelization semantics*: in program P_3, the

Figure 4: Illustration of how the `image_patch.find("drink")` call from Figure 1 can be adapted from a standard object detector (left) to a conformal one (right). For the latter, the green boxes are identified as definitely being in the image, whereas the yellow boxes may or may not be in the image.

statement `patch2.simple_query(...)` can be evaluated even though previous statements have not yet been evaluated, since all of its arguments (the image patch `patch2` and the string `"Does this have alcohol?"`) are constants. As a consequence, this external call can be dispatched in parallel with `patch1.simple_query(...)`, which significantly reduces execution time compared to sequential Python.

3.3 Conformal prediction

We implement *conformal semantics* (Ramalingam et al., 2024) in QUASAR for uncertainty quantification. When QUASAR makes external calls to other machine learning models, such as sub-agents, we may want to quantify the uncertainty of these models, and then keep track of how this uncertainty propagates through the program. Specifically, program variables are bound to sets of values instead of individual values.

The key challenge is modifying the program execution to handle sets of values. For example, if a Boolean variable x is bound to the set of values $x \mapsto \{\texttt{True}, \texttt{False}\}$, and there is a conditional statement **if** x `then` p_{true} **else** p_{false} (i.e., that branches on x), then we must execute both branches p_{true} and p_{false} of the conditional; then, for each variable y defined in these branches, we take the union of the values v_{true} bound to y in p_{true} and v_{false} bound to y in p_{false}, i.e., $y \mapsto v_{\text{true}} \cup v_{\text{false}}$. QUASAR includes a modified set of *conformal* rewrite rules that handle variables bound to sets of values in this way (see Appendix C).

Because external functions are opaque to QUASAR, abstract versions of them must be provided. In the case of calls to neural models, such as `find`, the abstract version is provided by applying a conformal technique, e.g., returning the set of labels whose probability is above some threshold. For example, the object detector shown on the left in Figure 4 misses two objects (though in this case, it does not affect the final answer in Figure 2); the output of the conformal detector is shown on the right. In this case, the external call `image_patch.find("drink")` indicates whether each detection is definitely (green) or possibly (yellow) in the image; it represents the set of lists of patches

$$\{[\texttt{patch1}, \texttt{patch2}, \texttt{patch3}, \texttt{patch4}], [\texttt{patch2}, \texttt{patch3}, \texttt{patch4}], [\texttt{patch1}, \texttt{patch2}, \texttt{patch4}], [\texttt{patch2}, \texttt{patch4}]\},$$

where the patches are ordered from left to right. Similarly, for each patch, the external call `patch.simple_query("Does this have alcohol?")` returns a prediction set that is a subset of $\{\texttt{"yes"}, \texttt{"no"}\}$. The result overapproximates the true output; in this case, the program output is $\{\texttt{"yes"}\}$, i.e., there is definitely an alcoholic drink in the image.

The conformal guarantee says that, for some target fraction of the test dataset ("coverage"), the ground truth label will be contained in the predicted set of labels. While this can be trivially obtained by outputting the set of all labels, the sets should be kept as small as possible while satisfying the target coverage. To satisfy the desired coverage guarantee, we use a standard conformal prediction strategy. First, we optimize the thresholds for each individual model on an optimization set (Li et al., 2024b). Then, using a held-out calibration set, we jointly rescale these thresholds using a single scaling parameter $\tau \in \mathbb{R}$ chosen using conformal prediction to satisfy a desired coverage guarantee (Angelopoulos et al., 2025;

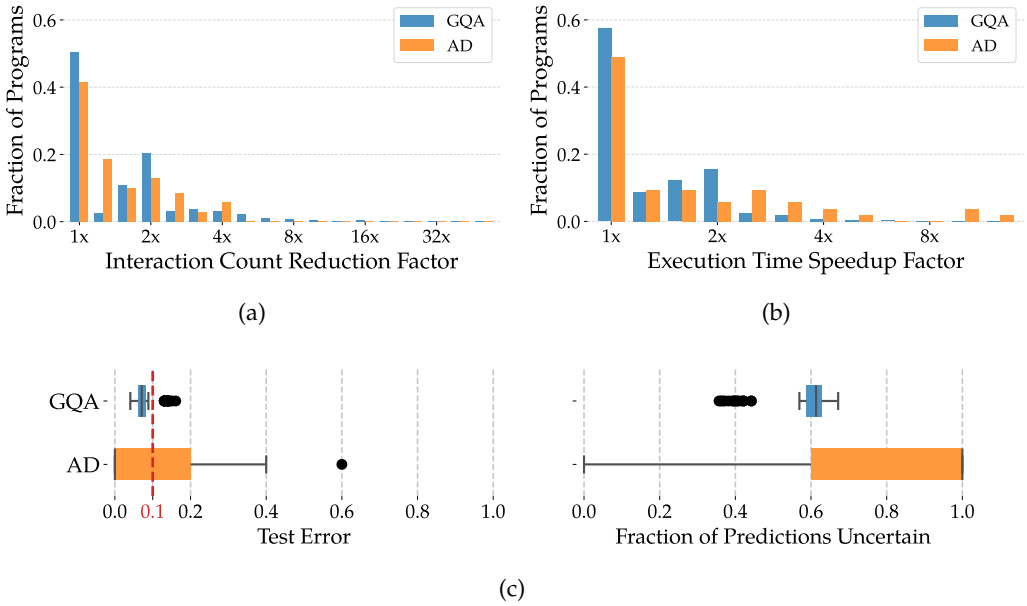


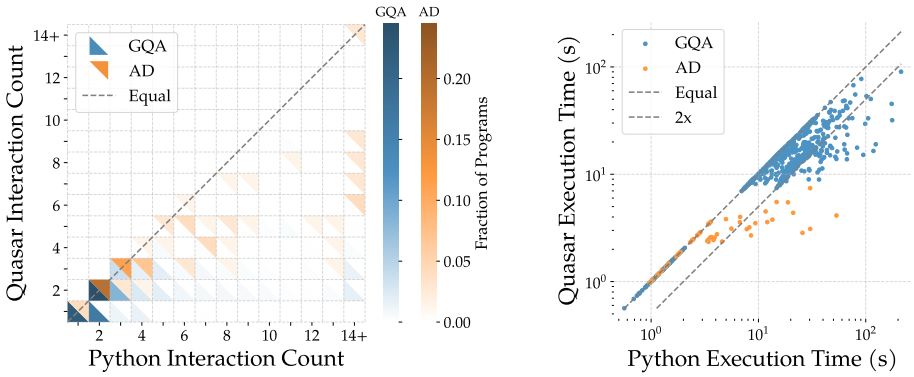
Figure 5: Distribution across tasks of interaction count reduction factors (Python count/QUASAR count) from approval batching (a). Distribution across tasks of execution time reduction factor (Python time/QUASAR time) from autoparallelization (b). Distributions across 100 validation/test splits of test error and uncertain prediction fraction for conformal semantics, satisfying a target 10% error rate (c). The target error rate is shown as a dashed line.

Zhang et al., 2025b). τ can also determine the number of programs to generate, in order to handle uncertainty at the program level (Quach et al., 2024).

4 Evaluation

We show that QUASAR significantly reduces the number of user interactions for access control, significantly reduces execution time, and achieves the target coverage rate using conformal prediction. We evaluate on ViperGPT (Surís et al., 2023), a visual question answering system, and CaMeL (Debenedetti et al., 2026), a secure AI assistant. Given a natural language query about an image, ViperGPT first uses an LLM to generate a Python program that would answer that query when provided with an image. The Python program itself has access to various neural modules, including an object detector, a vision-language model, and an LLM. We apply the ViperGPT approach to 1000 tasks randomly sampled from GQA (Hudson & Manning, 2019), a dataset of questions about various day-to-day images. The CaMeL AI assistant performs tasks for users by using an LLM to generate Python code that itself may invoke "quarantined" LLMs to process data without risk of prompt injections. We apply the CaMeL approach to all four suites of the AgentDojo (Debenedetti et al., 2024) benchmark. Unless otherwise stated, all of our results use programs generated by GPT-5 at the default ("Medium") reasoning level. We only include programs in our analysis that successfully compile and execute in QUASAR (i.e., no compilation or execution errors), but we include programs regardless of whether their output matches the ground truth label.

Access control. We evaluate the access control improvements in terms of the reduction in the number of user interactions required to approve all external calls, comparing the original Python program to its transpiled QUASAR counterpart. The transpiler we use guarantees that the Python source program and QUASAR target program make exactly the same external calls. The distribution of improvements is shown in Figure 5a and the interaction counts are shown in Figure 6a. On GQA, QUASAR reduces the number of user interactions by 26% $\pm$ 29



(a) (b)

Figure 6: Python vs. QUASAR user interactions required across tasks (a). Python vs. QUASAR execution time across tasks (b).

(mean $\pm$ stddev). The large variance is because only 50% of tasks allow approvals to be batched; among those, the number of interactions is reduced by 53% $\pm$ 17. On AgentDojo, the overall reduction is 26% $\pm$ 26, with 57% of tasks improvable and a reduction of 44% $\pm$ 19 on those. These results demonstrate that QUASAR can substantially reduce the frequency with which execution must block pending user approval.

Autoparallelization. We again consider pairs of Python programs and their transpiled QUASAR counterparts. To reduce variance in the results, we control for the time that each external call takes to execute, recording every external call that a program makes along with its result and execution time. Then, we replay this recording on both the Python and QUASAR versions of the program and record the total execution time of each. We exclude from the analysis programs that have negligible Python execution time ($\leq$ 100 ms). The distribution of improvements is shown in Figure 5b and the execution times are shown in Figure 6b. On the GQA dataset, QUASAR reduces running time by 18% $\pm$ 23. The large variance is because only 49% of tasks are parallelizable; among those, the running time is cut by 37% $\pm$ 19. On AgentDojo, the overall speedup is 27% $\pm$ 32, with 38% of tasks improvable and a speedup of 54% $\pm$ 22 on those. These results demonstrate that QUASAR can substantially reduce running time compared to Python execution.

Conformal semantics. We show that our conformal semantics can achieve a target error rate of 10% on a test set. For each benchmark, we divide the set of programs (one per task) into 100 random validation/test splits. For each split, we choose the largest threshold (and thus the smallest prediction sets) for which the validation error is below 10%, and then compute the test error at that threshold. The distribution of these test errors is shown in Figure 5c, with mean error rate 7.9% $\pm$ 2.8 for GQA and 8.6% $\pm$ 12.8 for AgentDojo. Because the domain of labels varies across tasks (e.g., yes/no, color, object, etc.), instead of measuring the size of prediction sets we measure certainty—i.e., the model is certain if the prediction set has size 1, and uncertain otherwise. We consider the fraction of tasks on which the model is uncertain. The distribution of such uncertainty rates is shown in Figure 5c, with mean 58.2% $\pm$ 8.3 in GQA and 84.1% $\pm$ 23.8 for AgentDojo. The high uncertainty rate for AgentDojo reflects the stringent certainty condition and the challenging nature of the dataset. These results demonstrate that QUASAR can achieve the desired coverage guarantee using conformal prediction.

Scaling to complex benchmarks. Finally, we perform an additional experiment to assess whether QUASAR can scale to more complex reasoning tasks. Specifically, we apply it to the BrowseComp-Plus (Chen et al., 2025) deep research dataset, which consists of question answering tasks where each task is accompanied by a large number ($\approx$ 100) of documents. The answer to each question requires integrating information across multiple documents; however, most documents are unrelated to the question and must be ignored. We apply the Recursive Language Models (RLM) (Zhang et al., 2025a) approach, adapted to use

QUASAR instead of Python. In RLM, rather than passing the documents directly into the LLM context, which may exceed the context window or lead to "context rot" (where model performance generally degrades as context size increases), the LLM is given access to a Python interpreter in which the documents exist as variables. Thus, it can manipulate documents programmatically and pass some subset of them to another LLM for processing, without needing to pollute its own context.

Due to the large number of model calls required for each task, we evaluate on a randomly selected subset of 80 BrowseComp-Plus tasks, using GPT-5-nano for the inner model calls. We see large improvements on this dataset: there is a 97% $\pm$ 1 reduction in the number of user approvals required and a 93% $\pm$ 4 speedup. Evaluating conformal prediction on this benchmark is complicated by the unstructured nature of the intermediate values and outputs, so we leave it to future work.

BrowseComp-Plus represents a realistic deep research benchmark, so these results demonstrate that QUASAR straightforwardly scales to complex tasks, with only the conformal prediction component requiring a domain-specific implementation. Indeed, in terms of latency and security improvements, we find stronger results on these complex tasks than on GQA and AgentDojo; complex tasks require comparatively more external calls, creating more opportunities for parallelization and batching.

5 Conclusion

We have presented QUASAR, a language for LLM code actions that distinguishes between internal code and external calls. Leveraging LLMs' proficiency with Python, we transpile from a subset of Python into QUASAR. Unlike Python, QUASAR has semantics that can be extended to support a wide variety of features. We demonstrate extensions for access control approval batching, automatic parallelization, and conformal prediction. Our experiments demonstrate that via these semantics, QUASAR can improve security, latency, and uncertainty quantification on both GQA and AgentDojo, and can furthermore scale to significantly more complex tasks such as BrowseComp-Plus.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF Awards CCF-1917852, CCF-2247088, CCF-2338777, and SLES 2331783, Amazon Research Award Fall 2023, and Amazon/ASSET Gift for Research in Trustworthy AI. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding entities.

Reproducibility Statement

A software artifact (Mell et al., 2026b) containing both the implementation and figure generation code is available at `https://github.com/stephenmell/quasar-colm2026-artifact`.

References

Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, Zico Kolter, Matt Fredrikson, Yarin Gal, and Xander Davies. AgentHarm: A benchmark for measuring harmfulness of LLM agents. In *International Conference on Learning Representations*, pp. 79185–79220, 2025.

Anastasios N. Angelopoulos, Stephen Bates, Emmanuel J. Candès, Michael I. Jordan, and Lihua Lei. Learn then test: Calibrating predictive algorithms to achieve risk control. *The Annals of Applied Statistics*, 19(2), 2025. doi: 10.1214/24-AOAS1998.

Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku, 2024. URL `https://www.anthropic.com/news/3-5-models-and-computer-use`.

Anthropic. Claude Code, 2026. URL https://claude.com/product/claude-code. AI-powered coding assistant for developers.

Adam Beautement, M. Angela Sasse, and Mike Wonham. The compliance budget: Managing security behaviour in organisations. In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW)*, pp. 47–58, 2008. doi: 10.1145/1595676.1595684. URL https://www.nspw.org/papers/2008/nspw2008-beautement.pdf.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446. URL https://arxiv.org/abs/2208.08227.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. BrowseComp-Plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, volume 37, pp. 82895–82920, 2024. doi: 10.52202/079017-2636.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. In *4th IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2026.

Feng He, Tianqing Zhu, Dayong Ye, Bo Liu, Wanlei Zhou, and Philip S. Yu. The emerged security and privacy of LLM agent: A survey with case studies. *ACM Computing Surveys*, 58(6):162:1–162:36, 2026. doi: 10.1145/3773080.

Drew A Hudson and Christopher D Manning. GQA: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6700–6709, 2019. doi: 10.1109/CVPR.2019.00686.

John Hughes. How we built Claude Code auto mode: a safer way to skip permissions. https://www.anthropic.com/engineering/claude-code-auto-mode, Mar 2026. Accessed: 2026-08-08.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474, 2020.

Shou Li, Andrey Kan, Laurent Callot, Bhavana Bhasker, Muhammad Shihab Rashid, and Timothy B Esler. REDO: Execution-free runtime error detection for COding agents, 2024a. URL https://arxiv.org/abs/2410.09117.

Shuo Li, Sangdon Park, Insup Lee, and Osbert Bastani. TRAQ: Trustworthy retrieval augmented question answering via conformal prediction. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3799–3821, 2024b. doi: 10.18653/v1/2024.naacl-long.210.

Linyu Liu, Yu Pan, Xiaocheng Li, and Guanting Chen. Uncertainty estimation and quantification for LLMs: A simple supervised approach, 2024a. URL https://arxiv.org/abs/2404.15993.

Na Liu, Liangyu Chen, Xiaoyu Tian, Wei Zou, Kaijiang Chen, and Ming Cui. From LLM to conversational agent: A memory enhanced architecture with fine-tuning of large language models, 2024b. URL https://arxiv.org/abs/2401.02777.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024. doi: 10.18653/v1/2024.acl-long.747.

Stephen Mell, Konstantinos Kallas, Steve Zdancewic, and Osbert Bastani. Opportunistically parallel lambda calculus. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025. doi: 10.1145/3763143.

Stephen Mell, David Mell, Konstantinos Kallas, Steve Zdancewic, and Osbert Bastani. PopPy: Opportunistically exploiting parallelism in Python compound AI applications, 2026a. URL https://arxiv.org/abs/2605.18697.

Stephen Mell, Botong Zhang, and David Mell. Artifact for "Quasar: A Programming Language Specialized for LLM Code Actions", August 2026b. URL https://doi.org/10.5281/zenodo.22086130.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. Measuring the impact of programming language distribution. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pp. 26619–26645, 2023. URL https://arxiv.org/abs/2302.01973.

Victor Quach, Adam Fisch, Tal Schuster, Adam Yala, Jae Ho Sohn, Tommi S Jaakkola, and Regina Barzilay. Conformal language modeling. In *International Conference on Learning Representations*, pp. 11654–11681, 2024.

Ramya Ramalingam, Sangdon Park, and Osbert Bastani. Uncertainty quantification for neurosymbolic programs via compositional conformal prediction, 2024. URL https://arxiv.org/abs/2405.15912.

James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. torch.fx: Practical program capture and transformation for deep learning in Python. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 4, pp. 638–651, 2022. URL https://arxiv.org/abs/2112.08429.

Ravi S. Sandhu. Role-based access control. In Marvin V. Zelkowitz (ed.), *Advances in Computers*, volume 46 of *Advances in Computers*, pp. 237–286. Elsevier, 1998. doi: 10.1016/S0065-2458(08)60206-5. URL https://www.sciencedirect.com/science/article/pii/S0065245808602065.

R.S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994. doi: 10.1109/35.312842.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551, 2023. doi: 10.52202/075280-2997.

Dídac Surís, Sachit Menon, and Carl Vondrick. ViperGPT: Visual inference via Python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 11854–11864, 2023. doi: 10.1109/ICCV51070.2023.01092.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024. doi: 10.18653/v1/2024.acl-long.850.

Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*, volume 29. Springer, 2005. doi: 10.1007/b106715.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022. doi: 10.52202/068431-1800.

Bin Wu. Introducing advanced tool use on the Claude Developer Platform. https://www.anthropic.com/engineering/advanced-tool-use, Nov 2025. Accessed: 2026-03-31.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822, 2023a. doi: 10.52202/075280-0517.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025a. URL https://arxiv.org/abs/2512.24601.

Botong Zhang, Shuo Li, and Osbert Bastani. Conformal structured prediction. In *International Conference on Learning Representations*, pp. 8563–8583, 2025b.

A Full QUASAR language

As described in Section 2, QUASAR executes programs by transforming them with rewrite rules until they reach a result. The syntax of programs is given in Figure 7. A program consists of a sequence of statements, where each statement defines some variable $(x, y, \ldots)$ to be the result of some operation (op). Variables are assumed to be defined exactly once (i.e., they are unique and do not shadow each other). An operation can be, in order: a primitive value c (where c ranges over Python values, such as `True`, `5`, or `"foo"`); another variable x; a tuple of variables x_i; the result of calling an external function f (from the set $\mathcal{F}_{\text{ext}}$) with argument x; the result of projecting out the i-th component from a tuple x; the result of folding over a list w with initial accumulator x and fold body $block$; an if expression on condition x with then-case $block_1$ and else-case $block_2$; or the result of some pending external call S. A block is a program that may additionally have some parameter x (so that the body of a fold can take the previous accumulator and the current list item as arguments). A value is either a Python object or a (possibly nested) tuple of Python objects—it does not directly occur in programs, but is used in the semantics.

The interpreter state at any time is simply a program P and a set E of dispatched external calls. The semantics consists of rewrite rules $R \in \mathcal{R}$, which transform one execution state into another, written $P, E \rightarrow_R P', E'$. Many rules do not affect E, and so are simply written as $P \rightarrow_R P'$. The rewrite rules are given in Figure 8. The rule "alias" removes a statement $y \leftarrow x$, replacing it with nothing, but renaming all occurrences of y in the program to x; "proj" replaces a projection operator, if the variable x is known to be a tuple $(x_1, \ldots, x_n)$, with the i-th element; for if statements, when the condition x is the primitive `True` ("if-t"), then the statement is replaced by a copy of $block_1$ (copying ensures that variables are unique; since blocks in if statements do not require parameters, w is bound to an empty tuple); if the condition is `False` ("if-f"), then the same is done for $block_2$; "fold" applies $block$ to each element of the list w, with x being the initial accumulator and y being the final one, and z_i being the i-th intermediate accumulator; "disp" replaces an external call to a function f when the argument x has a value $value$ (i.e., it is a primitive or a tuple of primitives, which $\text{value}(T, x)$ computes) with a placeholder S, begins executing the function f, and updates the execution set E; "ext" applies when an external function has finished executing—and so the execution set E contains a result in place of $\varnothing$—and replaces the placeholder with the result. In Section 2, we simplified S in the execution set to just be the external call statement itself, whereas here it is an identifier for the spawned task.

$$
\begin{aligned}
P &::= stmt_1; \ldots; stmt_n; \text{return } x \\
stmt &::= x \leftarrow op \\
op &::= \text{prim } c \\
&\mid x \\
&\mid (x_1, \ldots, x_n) \\
&\mid f\ x \\
&\mid \text{proj } i\ x \\
&\mid \text{fold } w\ x\ block \\
&\mid \text{if } x\ block_1\ block_2 \\
&\mid ?S \\
block &::= \{x \Rightarrow P\} \\
value &::= c \mid (value_1, \ldots, value_n)
\end{aligned}
$$

Figure 7: The grammar defining programs in QUASAR.

$$\frac{}{T[y \leftarrow x] \rightarrow \text{rename}(T[\![\varnothing]\!], y, x)}\text{(alias)} \qquad \frac{(x \leftarrow (x_1, \ldots, x_n)) \in T}{T[y \leftarrow \text{proj } i\ x] \rightarrow T[\![y \leftarrow x_i]\!]}\text{(proj)}$$

$$\frac{(x \leftarrow \text{prim } \texttt{True}) \in T \qquad \{w \Rightarrow stmts; \text{return } z\} = \text{copy}(block_1)}{T[y \leftarrow \text{if } x\ block_1\ block_2] \rightarrow T[\![w \leftarrow (); stmts; y \leftarrow z]\!]}\text{(if-t)}$$

$$\frac{(x \leftarrow \text{prim } \texttt{False}) \in T \qquad \{w \Rightarrow stmts; \text{return } z\} = \text{copy}(block_2)}{T[y \leftarrow \text{if } x\ block_1\ block_2] \rightarrow T[\![w \leftarrow (); stmts; y \leftarrow z]\!]}\text{(if-f)}$$

$$\frac{\begin{array}{c}(w \leftarrow \text{prim } [c_1, \ldots, c_n]) \in T \\ \forall i. \{y_i \Rightarrow stmts_i; \text{return } z_i\} = \text{copy}(block) \qquad stmts'_i = (w_i \leftarrow \text{prim } c_i; y_i \leftarrow (z_{i-1}, w_i); stmts_i)\end{array}}{T[y \leftarrow \text{fold } w\ x\ block] \rightarrow T[\![z_0 \leftarrow x; stmts'_1; \ldots; stmts'_n; y \leftarrow z_n]\!]}\text{(fold)}$$

$$\frac{\text{value}(T, x) = value \qquad S = \text{spawn}(f, value)}{T[y \leftarrow f\ x], E \rightarrow T[\![y \leftarrow ?S]\!], E \cup \{(S, \varnothing)\}}\text{(disp)}$$

$$\frac{term = (stmts; \text{return } x)}{T[y \leftarrow ?S], E \cup \{(S, term)\} \rightarrow T[\![stmts; y \leftarrow x]\!], E}\text{(ext)}$$

(a)

$$\frac{(x \leftarrow \text{prim } c) \in T}{\text{value}(T, x) = c} \qquad \frac{(x \leftarrow (x_1, \ldots, x_n)) \in T \qquad \forall i. \text{value}(T, x_i) = value_i}{\text{value}(T, x) = (value_1, \ldots, value_n)}$$

(b)

Figure 8: The rewrite rules of the semantics of QUASAR (a), and the formal definition of the value function used by the "disp" rule (b). $T[stmt]$ means a program P with some statement $stmt$ in it; $T[\![stmts]\!]$ means that the statement $stmt$ was replaced by the list of statements $stmts$. For each rule, the $P, E \rightarrow P', E'$ below the line is an allowed rewrite, subject to all of the conditions above the line. If E is not modified by a rule, we omit it for concision. $stmt \in T$ means that $stmt$ is in the list of statements of T.

B Transpilation

QUASAR is functional, while Python is imperative. Thus, in QUASAR, variables cannot be changed once they have been defined. Being functional makes supporting parallel, partial, and conformal execution possible, but the Python code generated by agents performs imperative updates to local variables. Handling updates in "straight-line" code (without control-flow structures like `if` and `for`) is straightforward. However, updates inside control-flow structures, which therefore may or may not happen, are more challenging. In the example from Figure 2, `found` may be updated inside the loop.

We adapt an existing transpiler (Mell et al., 2026a) to translate to QUASAR an expressive subset of Python that includes function calls, local variable assignments, and the `if` and `for` control-flow constructs. To support imperative control-flow structures in Python, we convert them into a functional form. `if` statements in Python are transformed as shown in Figure 9, where variables that might be updated by a statement-level conditional are instead returned from an expression-level conditional. A similar translation is done from imperative `for` loops to functional fold operations: variables that might be updated by the `for` loop are instead passed as the fold accumulator (in Python, fold is called `reduce`).

```
found = False
cond = drink_patch.simple_query("...")
if cond:
    found = True
```

(a)

$\rightsquigarrow$

```
found0 = False
cond = drink_patch.simple_query("...")
def then_case():
    found1 = True
    return found1
def else_case():
    return found0
found2 = then_case() if cond else else_case()
```

(b)

Figure 9: An illustration of the translation of "if" statements, shown in Python syntax. Before, with imperative updates in "if" statements (a). After, with no imperative updates and a functional conditional operation (b).

C Conformal semantics for QUASAR

To support conformal evaluation, QUASAR must be extended to handle sets of values. The syntax has three additional operations, as shown in Figure 10. In order: abstract primitives represent the set of Python values c_i; abstract lists represent a list where some of the elements may be uncertain, in that if b_i is **False**, the element c_i may or may not be in the list, whereas if b_i is **True**, then c_i is definitely in the list; and join operations combine multiple computations into a set. Join is distinct from an abstract set, since in the latter the values must be known, whereas in the former they may not yet be computed.

The semantics also contains additional rules to support these new operations, as shown in Figure 11. The rule "join-join" applies when x is the join of variables, and one of them, y, is itself a join, in which case they can be flattened to a single join; "join-tuple" applies when x is the join of n tuples of identical length m, in which case it becomes the tuple of joins of the respective components; "join-prim" applies when x is the join of n primitives c_i, in which case it becomes an abstract set of those values; "if-tf" applies when the condition of an if statement is the abstract set of both **True** and **False**, in which case both branches are taken, resulting in z_1 and z_2, which are joined to produce y; "fold-abs" applies when folding over an abstract list, in which case a copy of *block* is made for each element of the list; however, if a list element is uncertain ($b_i =$ **False**), then the resulting accumulator z_i is joined with z_{i-1} to capture both the case where c_i is in the list and the case where it is not.

$$
\begin{aligned}
op &::= \ldots \\
&\mid \text{absprim } \{c_1, \ldots, c_n\} \\
&\mid \text{abslist } [(c_1, b_1), \ldots, (c_n, b_n)] \\
&\mid \text{join } \{x_1, \ldots, x_n\} \\
\mathit{absvalue} &::= \{c_1, \ldots, c_n\} \mid (\mathit{absvalue}_1, \ldots, \mathit{absvalue}_n)
\end{aligned}
$$

Figure 10: The additional operations in the grammar of QUASAR to support conformal evaluation.

$$
\frac{(y \leftarrow \text{join } \{y_1, \ldots, y_m\}) \in T}{T[x \leftarrow \text{join } \{x_1, \ldots, x_n, y\}] \rightarrow T[\![x \leftarrow \text{join } \{x_1, \ldots, x_n, y_1, \ldots, y_m\}]\!]} \text{(join-join)}
$$

$$
\frac{\forall i.(x_i \leftarrow (w_{i,1}, \ldots, w_{i,m})) \in T \qquad \mathit{stmts}_j = (y_j \leftarrow \text{join } \{w_{1,j}, \ldots, w_{n,j}\})}{T[x \leftarrow \text{join } \{x_1, \ldots, x_n\}] \rightarrow T[\![\mathit{stmts}_1; \ldots; \mathit{stmts}_m; x \leftarrow (y_1, \ldots, y_m)]\!]} \text{(join-tuple)}
$$

$$
\frac{\forall i.(x_i \leftarrow \text{prim } c_i) \in T}{T[x \leftarrow \text{join } \{x_1, \ldots, x_n\}] \rightarrow T[\![x \leftarrow \text{absprim } \{c_1, \ldots, c_n\}]\!]} \text{(join-prim)}
$$

$$
\frac{\begin{gathered}(x \leftarrow \text{absprim } \{\texttt{True}, \texttt{False}\}) \in T \\ \{w_1 \Rightarrow \mathit{stmts}_1; \text{return } z_1\} = \text{copy}(\mathit{block}_1) \qquad \{w_2 \Rightarrow \mathit{stmts}_2; \text{return } z_2\} = \text{copy}(\mathit{block}_2)\end{gathered}}{T[y \leftarrow \text{if } x \ \mathit{block}_1 \ \mathit{block}_2] \rightarrow T[\![w_1 \leftarrow (); \mathit{stmts}_1; w_2 \leftarrow (); \mathit{stmts}_2; y \leftarrow \text{join } \{z_1, z_2\}]\!]} \text{(if-tf)}
$$

$$
\frac{\begin{gathered}(w \leftarrow \text{abslist } [(c_1, b_1), \ldots, (c_n, b_n)]) \in T \qquad \forall i.\{y_i \Rightarrow \mathit{stmts}_i; \text{return } z_i\} = \text{copy}(\mathit{block}) \\ \mathit{stmts}'_i = (w_i \leftarrow \text{prim } c_i; y_i \leftarrow (z_{i-1}, w_i); \mathit{stmts}_i) \\ \mathit{stmts}''_i = \text{if } b_i \text{ then } \mathit{stmts}'_i \text{ else } (\mathit{stmts}'_i; z_i \leftarrow \text{join } \{z_i, z_{i-1}\})\end{gathered}}{T[y \leftarrow \text{fold } w \ x \ \mathit{block}] \rightarrow T[\![z_0 \leftarrow x; \mathit{stmts}''_1; \ldots; \mathit{stmts}''_n; y \leftarrow z_n]\!]} \text{(fold-abs)}
$$

Figure 11: The additional rewrite rules in the semantics of QUASAR to support conformal evaluation.