EasyTracker: A Python Library for Controlling and Inspecting Program Execution

Théo Barollet, Christophe Guillon, Manuel Selva, François Broquedis, Florent Bouchez-Tichadou, Fabrice Rastello

To cite this version:

Théo Barollet, Christophe Guillon, Manuel Selva, François Broquedis, Florent Bouchez-Tichadou, et al.. EasyTracker: A Python Library for Controlling and Inspecting Program Execution. CGO 2024 - International Symposium on Code Generation and Optimization, Mar 2024, Edinburgh, United Kingdom. pp.1-14. hal-04368835v3

HAL Id: hal-04368835

https://inria.hal.science/hal-04368835v3

Submitted on 13 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

EasyTracker: A Python Library for Controlling and Inspecting Program Execution

Théo Barollet
Univ. Grenoble Alpes
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
theo.barollet@inria.fr

Christophe Guillon
Univ. Grenoble Alpes
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
christophe.guillon@inria.fr

Manuel Selva
Univ. Grenoble Alpes
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
manuel.selva@inria.fr

François Broquedis
Univ. Grenoble Alpes
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
francois.broquedis@inria.fr

Florent Bouchez-Tichadou
Univ. Grenoble Alpes
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
florent.bouchez-tichadou@inria.fr

Fabrice Rastello
Univ. Grenoble Alpes
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
fabrice.rastello@inria.fr

***Abstract*—Learning to program involves building a mental representation of how a machine executes instructions and stores data in memory. To help students, teachers often use visual representations to illustrate the execution of programs or particular concepts in their lectures. As a famous example, teachers often represent references/pointers with arrows pointing to objects or memory locations. While these visual representations are mostly hand-drawn, there is a tendency to supplement them with tools.**

However, building such a tool from scratch requires much effort and a high level of debugging technical expertise, while existing tools are difficult to adapt to different contexts. This article presents EasyTracker, a Python library targeting teachers who are not debugging experts. By providing ways of controlling the execution and inspecting the state of programs, EasyTracker simplifies the development of tools that generate tuned visual representations from the controlled execution of a program. The controlled program can be written either in Python, C, or assembly languages. To ease the development of visualization tools working for programs in different languages and to allow the building of web-based tools, EasyTracker provides a language-agnostic and serializable representation of the state of a running program.

***Index Terms*—teaching, visualization, debug, instrumentation**

I. Introduction

Computer science teachers use visual representations extensively to illustrate their lectures, especially when teaching programming. Such representations may be either hand-drawn or automatically generated.

While we can all acknowledge the positive impact of hand-drawn representations on the learning process, we believe that generated representations are helpful for:

- answering “what if questions” asked in class with live demonstrations;
- generating images and videos for the material complementing/replacing lectures;
- visualizing real-size problems;
- empowering learners to self-validate their understanding of the concept highlighted in the representation. For example, learners can compare the representation generated by a visualization tool with the one they have in mind without the teacher’s intervention.

The success of Python Tutor (PT) with its 10M users in the last decade (1) supports the claim that generated representations are useful.

Unfortunately, no generic tool can fit all the specific visualization needs of all particular teacher and student audiences. As an illustration, when we started this project, we had many diverse scenarios in mind, including:

- the teaching of algorithms where one wants to show to the students the invariants on high level data structures as the program executes;
- the teaching of languages where one wants to show important notions such as scopes, pointers and stack frames;
- the teaching of debugging through a serious game, with interactive and rewarding visualizations of the program state;
- the teaching, during our compiler courses, of basic architecture where one wants to show the raw memory along with the hardware registers, particularly the program counter and the stack pointer.

In all these scenarios, a fundamental objective was to let students modify their programs and visualize the effect of their changes. Clearly, a debugger corresponds to the appropriate infrastructure for interacting with programs. However, our observation is that implementing tools that automatically generate custom visual representations currently requires quite a high level of programming commitment from teachers and a strong debugging expertise. In other words, if existing generic tools such as PT do not fulfill a teacher’s need, one must build a tool from scratch.

In this work, we present a Python library called EasyTracker designed to assist teachers in building tools capable of

```python
#!/usr/bin/env python3
"""Sorting algorithm to illustrate the notion of loop invariants"""


def sort(array):
    """Sorts array containing only ones and zeros"""
    i = 0
    j = len(array) - 1
    while i != j:
        if array[i] == 0:
            i += 1
        else:
            array[i] = array[j]
            array[j] = 1
            j -= 1


def main():
    """Tests the above function"""
    array = [0, 1, 0, 1, 0, 1, 0, 1, 0]
    print("before :", array)
    sort(array)
    print("after :", array)


if __name__ == "__main__":
    main()
```

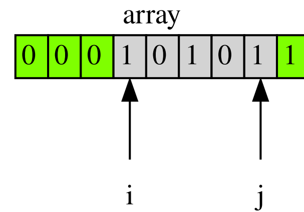

Fig. 1: A visualization tool built on top of EasyTracker to visually represent the concept of loop invariant. The next line to execute is highlighted in the source code and the loop invariant visualization exhibits the principle of the algorithm.

generating visual representations of running programs written in Python, C, or assembly. We believe that the development of a program visualization tool is made of two parts: the first for controlling and inspecting the executing program and the second for visualizing and interacting. The goal of EasyTracker is to decouple those two components and provide an interface for the first part. We should also clarify that the primary goal of EasyTracker is not to simplify the development of debugger GUIs, which, as illustrated later in Section IV, rarely meet teachers' requirements. Furthermore, it is essential to note that EasyTracker does not replace libraries providing access to GDB (2) in one's favorite programming language such as pygdbmi (3) for Python or gdbwire (4) for C. Instead, EasyTracker is developed on top of these low-level libraries, providing a higher level abstraction that is more accessible to *non-expert teachers*.

Figure 1 depicts a visualization tool we built on top of EasyTracker to visually represent the concept of loop invariants. This tool shows the source code of the program under its control (top-left of the figure) along with the state of the array while it is sorted (right of the figure). The visualized program is executed line by line when striking enter. Invariants are visually represented by the `i` and `j` indices and by the fact that elements already sorted are highlighted with a darker background color in the array.

Figure 2 shows the global architecture of a visualization tool built on top of EasyTracker, such as the one above. The visualization tool must be written by the teacher, for instance, using some existing Python visualization library. It uses the EasyTracker API to control the execution of a program referred to as the *inferior*[1]. Depending on the context, the teacher will need a tool that provides a way to control the execution through the keyboard or buttons or a tool that executes the entire program to generate an image of the program state after the execution of each line of the inferior source code. Any time the inferior is *paused*, the tool can inspect its state using EasyTracker. This state includes where the program is paused, what is the current stack and what are the live variables along with their values. The tool then updates its graphical user interface according to the result of the state inspection.

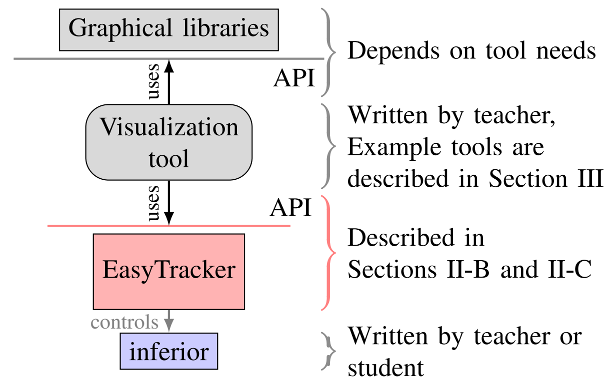

Fig. 2: A visualization tool is built on top of the control and inspection features of EasyTracker and uses graphical libraries.

As further detailed, compared to a debugger, EasyTracker provides: 1. a simple/higher level *programmatic* interface; 2. a *serializable abstract* representation of program state; 3. both common to Python, C and assembly inferiors (language-*agnostic*).

Section II presents the details of the EasyTracker API along with its current implementations for C and assembly programs on one side and Python programs on the other. As a first evaluation of the EasyTracker APIs expressiveness, Section III presents four tools built on top of EasyTracker we used in our classes. Section IV presents related work before we conclude in Section V.

[1] this terminology comes from GDB where it refers to the program being debugged

II. The EasyTracker Library

The main objective of this work is to provide an open-source library [2] facilitating the development of program visualization tools for programming courses.

A. Main Requirements

The main requirements/features are the following:

- run, control and inspect an executable program [3];
- describe the inspection and control using imperative-style scripts;
- provide a language-agnostic API;
- be as high level and simple as possible.

The *control* API should allow to pause (and resume) the execution at control points by specifying/restricting to:

- the entry/exit of function calls;
- call depth;
- triggered event such as variable modification;
- stepping at line granularity.

Once the execution is paused, the *inspection* API should allow to inspect both local and global variables either:

- individually, by looking up from its name and the function name;
- or exhaustively, by walking through the stack-frame
- providing their scope, type, and value (including references)

EasyTracker fulfills those requirements and comes with two implementations that we call *trackers*: the first one dedicated to inferiors written in C or in assembly, and the second dedicated to Python inferiors.

B. The EasyTracker Interfaces

Before providing the details of our interface, let us consider the script given in Listing 1 that implements the stack-and-heap visualization tool described in Section III (Fig. 6(c)). After executing each line, this tool stalls the program and generates one image representing both stack, globals and heap current states.

```
1  inf = sys.argv[1]
2  tracker = init_tracker("python" if inf.endswith(".py")
3                         else "GDB")
4  tracker.load_program(inf)
5  tracker.start()
6  img_count = 1
7  while tracker.get_exit_code() is None:
8      frame = tracker.get_current_frame()
9      draw_stack_heap(frame, f"img{img_count}.svg")
10     tracker.step()
11     img_count += 1
12
13 def draw_stack_heap(frame: Frame, img_name: str) -> None:
14   frms_2_vars: dict[tuple[str, int], list[Variable]] = {}
15   while frame is not None:
16       variables = list(frame.variables.values())
17       frms_2_vars[(frame.name, frame.depth)] = variables
18       frame = frame.parent
19   draw_stack(frms_2_vars, img_name)
20   draw_heap(frms_2_vars, img_name)
```

Listing 1: Code of our language-agnostic stack-and-heap visualization tool that steps through the program and generates one image per executed line.

```
def break_before_line(lineno: int,
                      maxdepth: int=infty) -> None:
  """pauses the inferior before executing line lineno"""
def break_before_func(func: str,
                      maxdepth: int=infty) -> None:
  """pauses the inferior when entering func"""
def watch(variable:str, func_name: str=None) -> None:
  """pauses the inferior every time the value of
     the variable referenced by variableId changes"""
def track_function(func: str,
                   maxdepth: int=infty) -> None:
  """pauses the inferior when entering/exiting func"""
```

Listing 2: Functions to indicate when to pause the inferior.

Here, `inf` (defined on line 1) is the name of the inferior that the tool will execute. As already mentioned, EasyTracker currently comes with two trackers (Python and GDB) and line 2 sets the tracker to use. The tool then loads the inferior on line 4 and starts executing it on line 5. The control loop on line 7 is typical of many EasyTracker tools. In this example, the tool steps through every line of the inferior on line 10, but it could define higher level pause conditions such as watching a variable and calling the resume function (see e.g. Listing 6). The local variables' state is gathered thanks to the `get_current_frame` function on line 8 and inspected by the `draw_stack_heap` function.

1) The Control Interface: The control interface provides the functions shown in Listing 2 to tell EasyTracker when to pause the execution of the inferior.

[2]source-code: https://gitlab.inria.fr/corse/easytracker and documentation: https://corse.gitlabpages.inria.fr/easytracker/

[3]Note that an interpretable trace can be seen as a deterministic executable

The `break_before_line` and `break_before_func` functions inform EasyTracker that the inferior must be paused just before executing a given line or just before entering a given function. Returning from `break_before_func` guarantees the arguments are initialized, hence accessible, when the inferior is paused.

`track_function` informs EasyTracker that the inferior must be paused at the beginning (just after entering) and at the end (just before returning) of every execution of `funcname`. `watch` makes EasyTracker pause the inferior every time the variable identified by `variableId` is modified.

One can use the `maxdepth` optional parameter to tell EasyTracker to pause the inferior only if the current frame depth is below a given value.

Like a debugger, the control interface also provides functions to start/resume the execution of the inferior as described in Listing 3.

Each of these functions returns an instance of `PauseReason`, a class representing why EasyTracker paused the inferior program. We set a priority for each of the possible pause reasons so the functions above return the condition with the highest priority that triggered the pause. Here is a list of the possible pause reasons, sorted by priority in descending order : 1) The inferior exited. 2) A watched

```python
def start() -> PauseReason:
    """starts the inferior and immediately pauses it"""
def next() -> PauseReason:
    """executes one line without jumping into functions"""
def step() -> PauseReason:
    """executes one line with jumping into functions"""
def resume() -> PauseReason:
    """resumes until a pause condition has been reached"""
```

Listing 3: Functions to start and to resume the inferior.

variable has been modified, or we have reached the boundary of a tracked function. 3) A tracked function has been entered or exited. 4) A line breakpoint has been hit. 5) The end of a single-stepping control command (`start`, `next` or `step`) has been reached.

Again, we emphasize on the fact that functions of the control interface return only when the inferior is paused or terminated.

2) The Inspection Interface: This interface defines how a tool can observe the current state of a paused inferior program.

a) The functions: Listing 4 describes the functions called to know where in the source code the inferior has been paused. Listing 5 lists functions that users can call to get frames, global variables, register values or to recover the inferior exit code. `get_registers_gdb` and `get_value_at_gdb` functions are specific to the GDB tracker (ref to Section II-C).

```python
def get_last_lineno() -> int:
    """returns the number of the last executed line"""
def get_next_lineno() -> int:
    """returns the number of the next line to execute"""
```

Listing 4: Functions related to the inferior source code.

```python
def get_exit_code() -> Optional[int]:
    """returns inferior's exit code or None if running"""
def get_current_frame() -> Frame:
    """returns the innermost Frame (the deepest one)"""
def get_global_variables() -> dict[str, Variable]:
    """returns a list of Value for global variables"""
def get_variable_value(name: str) -> Optional[Variable]:
    """returns the given variable value if in scope"""
def get_registers_gdb() -> List[Register]:
    """returns a list of registers (name + value)"""
def get_value_at_gdb(address: int) -> int:
    """returns the value in memory at the given address"""
```

Listing 5: Functions to inspect frames, variables and to get the inferior exit code.

b) The representation of the program's state: Figure 3 shows the class diagram for the serializable representation of the state of a paused program. The `Frame` and `Variable` classes are self explanatory. The heart of the representation is then the `Value` class, which is the type of the `value` attribute of a `Variable` instance.

The `abstract_type` attribute indicates the nature of the `Value` and then what is found in the `content` attribute:

- `PRIMITIVE` represents Python's int, float and str and C int, long, double, float, char and char*. In this case, the `content` attribute contains the associated Python's primitive types.
- `REF` represents C pointers, Python variables and class attributes. In this case, the `content` attribute is a `Value`.
- `LIST` represents C arrays and Python lists and tuples. In this case, the `content` attribute is a tuple (chosen over a list for immutability) of `Value`.
- `DICT` represents Python dictionaries. In this case, the `content` is a dictionary from `Value` to `Value`.
- `STRUCT` represents C structures and all Python instances not fitting in one of the other abstract types. In this case, the `content` is a dictionary from `str` to `Value`.
- `NONE` represents Python `None` instance. In this case, the `content` attribute is `None`.
- `INVALID` represents C invalid pointers. In this case, the `content` attribute is `None`.
- `FUNCTION` represents C function pointers and Python functions. In this case, the `content` attribute is an `str` being the name of the function.

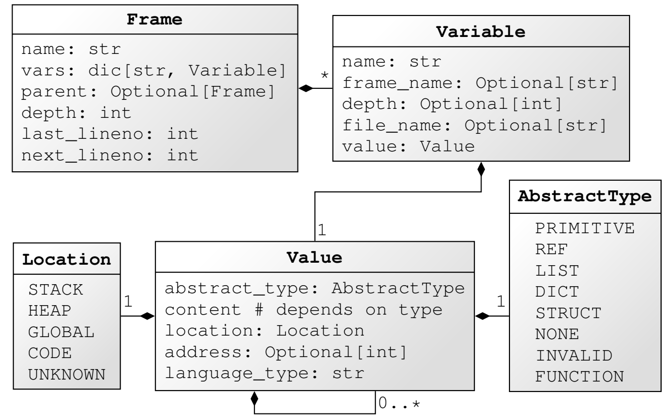

Fig. 3: Class diagram for the serializable representation of the state of a paused program written either in Python or in C.

The `location` attribute of a `Value` indicates where it lies in the conceptual memory of the program. By conceptual we refer for example to the fact that all Python variables have a `REF` `Value` in the stack pointing to the heap.

The `address` attributes indicates where exactly a `Value` lies in memory. For Python `instances` we use the `id` function (returning the object's memory address in CPython, the most used Python's implementation) to fill the `address` of all `Value` instances except for `REF` ones for which the notion of address makes no sense.

The `language_type` attributes is an `str` representation of the type in the inferior language terminology. For example it is `"char*"` for a `PRIMITIVE` representing a string in C or `"tuple"` for a `LIST` representing a Python tuple.

C. Python and GDB Based Implementations

EasyTracker comes with two implementations of the interface described in the previous section. One is for tracking Python code, and the other can track a program written in C, assembly

and virtually all compiled languages supported by GDB (the GNU Debugger). We wrote both these implementations in Python.

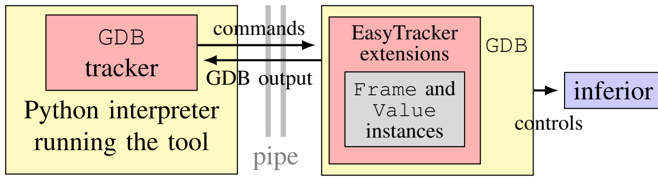

Fig. 4: The GDB tracker: yellow boxes are processes and red ones are the tracker implementation.

1) The GDB Tracker for C and Assembly Programs: As the name suggests, the GDB tracker is based on GDB and relies on custom extensions we provide, as shown in the right part of Fig. 4. The GDB tracker runs GDB as a subprocess in Machine Interface mode (MI). The former interacts with the latter through a pipe. To send a command to GDB, the tracker writes the command to the pipe.

GDB already provides almost all the control commands the EasyTracker control interface requires. Implementing these commands in the tracker then boils down to calling the proper function from the GBD MI interface. Nevertheless, two features are missing in GDB for implementing the EasyTracker control interface.

The first one concerns the `maxdepth` semantic described in the previous section. We implemented it as a Python-based GDB extension, adding custom breakpoints commands that take an additional `maxdepth` parameter. Every time such a custom breakpoint is hit, the GDB tracker checks if the current frame depth is deeper in the stack than its `maxdepth` argument. If it is, then the tracker simply resumes the execution of the inferior.

The second missing feature in GDB is the `track_function` functionality. Indeed, GDB allows placing a breakpoint at the function entry but not at function exit. GDB provides a `finish` command that stops when the current function returns but this does not place a breakpoint, so if the program is interrupted on the way it will not stop later when the program reaches the end of the function. We also cannot place a breakpoint just after the call because we still want to access frame information and variable values so a breakpoint has to be placed inside the function. To solve this issue, we use the disassembly feature of GDB along with a breakpoint on function entry. When we enter a tracked function for the first time, we disassemble the function code and look for the `retq x86` instruction that returns from the function. We can place a breakpoint at the address of this instruction. This works in many cases since it is a common practice in compiler designs to write a single function epilogue and thus a single `retq` instruction. This implementation is thus currently restricted to `x86` architectures. Nevertheless we could implement the same mechanism and choose the condition to find the return instruction depending on the actual architecture.

Regarding the implementation of the inspection interface, GDB only provides some printing ability. Consequently, we extended it with a custom inspection command based on the GDB `backtrace` command. Our custom inspection command recursively explores stack frames and the memory locations accessible from local variables to create `Frame`, `Variables` and `Value` instances as described in Section II-B2. To that end, we again use the GDB Python interface but this time to access each memory location's type and value.

Moreover, to implement the inspection interface EasyTracker needs to know the size of dynamic memory allocations on the heap. Indeed, if an integer array is heap-allocated GDB only knows the `int*` type but not the array size. To solve this issue, we wrote a thin library to override the dynamic allocation functions `malloc`, `free`, `calloc`, and `realloc`. Our custom allocation functions simply call the corresponding functions in the standard library, but before returning, they assign their arguments and return values to local variables. This allows us to add internal watchpoints on these variables. Every time such a watchpoint is hit, we silently update a list of heap-allocated blocks and their size and resume the execution of the inferior. As a consequence the GDB tracker knows if pointer values refer to heap-allocated blocks or not and if it is the case, it knows the corresponding size. Such overrides on the C runtime are automatically loaded on `load_program` thanks to the dynamic loader `LD_PRELOAD` feature.

The `Frame`, `Variables` and `Value` instances live inside the memory of a Python interpreter embedded into GDB as shown in Fig. 4. They then need to be transferred through the pipe to the memory of the Python interpreter running the tool. For that, our GDB extension serializes the instances and then writes them on the pipe, and the tracker deserializes them on the other end. Because both sides manipulate Python objects, we can use the standard serialization mechanism of Python.

2) The Python Tracker: Unlike the GDB tracker, the Python tracker runs in the same process as the inferior, significantly simplifying the tracker's inspection part. The Python standard library offers quite an elaborated and easy-to-use memory inspection interface through the `inspect` module. Here, besides implementing the complex class hierarchy described in section II-B2, the `Variable` instances can directly encapsulate the Python object they represent. This optimization results in an extended API for tool designers targeting Python programs, which, although available in EasyTracker, will not be detailed here for the sake of simplicity.

The hard job of the Python tracker is then to control the execution of the inferior. Python comes with a basic extendable debugger called `bdb`. However, `bdb` does not support watchpoints, and the function tracking features are not straightforwardly implementable in this debugger. Consequently, we decided to implement the Python tracker directly on top of the `sys.settrace` functionality that `bdb` internally uses. This function allows to register a trace function called by the interpreter, among other things, just before executing a line of Python source code. The trace function has three parameters: the current frame, the type of the line to be executed (line,

function call, function return) and possibly the return value if the line is a function return.

To implement watchpoints on top of `sys.settrace`, we check before the execution of every line if the value of any watched variable has changed. Consequently, even when the inferior is resumed through a call to `resume`, single-stepping line by line is done to determine whether EasyTracker should pause the inferior. Note that this slows the execution down a lot. However, it is not critical for the pedagogical context we target.

Implementing function tracking in the Python tracker is more accessible than in the one we developed for GDB-supported languages. Indeed, the interpreter calls our trace function after entering a new function and before exiting a function.

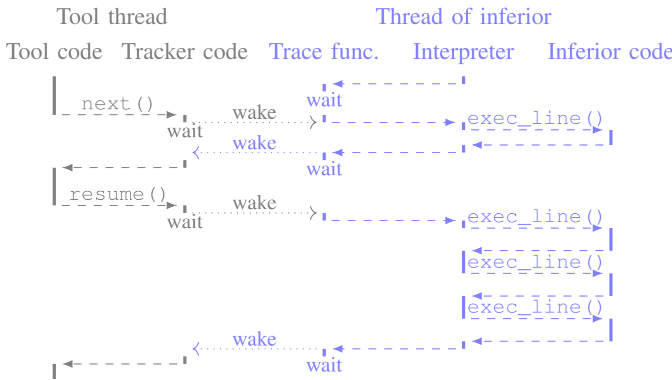

Fig. 5: The Python tracker runs the inferior in a thread.

Any Python debugger based on `sys.settrace` runs in the same process as the inferior it controls. In our case, this process is the Python interpreter running the tool. As EasyTracker imposes a call to a function from the control interface should not return until the execution of the inferior pauses, the execution of the inferior must be decoupled from the execution of the tool's code.

We decided to go for a thread-based implementation as shown on Fig. 5 to implement this decoupling. We call the *tool thread* the main thread of the Python interpreter executing the tool's code. EasyTracker executes the inferior in a dedicated thread so it can be paused. To have a better understanding of the implementation, Fig. 5 shows which thread executes which piece of code. The tool thread executes either the code of the tool or the code of the tracker. The inferior thread executes the inferior code as the name suggests. This thread also executes the trace function code registered with `sys.settrace` between the execution of every single line of code of the inferior. Thus, EasyTracker performs the control of the inferior from inside the code of the trace function. Also, it is clear from the diagram that the tool thread waits for the inferior to pause again after calling a control function.

Compared to the GDB implementation, the inferior runs in a thread and not a process, both the tool and the inferior run inside the same Python interpreter. As already stated, this makes introspecting the value of the inferior variables straightforward as both the tracker and the tool can directly access these values.

III. Tools Built On Top of EasyTracker

To demonstrate what can be achieved with EasyTracker, we describe in this section our own tools we built on top of it.

A. Python/C Stack and StackHeap Diagrams

We used EasyTracker to automatically generate stack (5) and stack-and-heap (6) diagrams that we use in our Python and C programming courses materials. This tool takes as input a Python or a C program along with display options and generates either a stack diagram or a stack-and-heap diagram after the execution of every line. The source code of the tool is the one already shown in Listing 1 and we can see the result of this tool on two different Python programs in Fig. 6.

The first course that focuses on making students understand stack frame uses the stack diagram of Fig. 6(a). The concept of references along with the emphasis that every variable is a reference in Python are introduced later by augmenting it with the heap as shown in Fig. 6(b). A generic tool such as PT does not allow to only show the stack with inlined values for both primitive types and all other types such as `list` and `tuples`.

Fig. 6(c) shows the result of the stack-and-heap tool on a C program. We represent invalid pointers with a cross. This example is used to show how C compares to Python: here the value of a variable can be in the stack, and we can have pointers targeting the stack.

One should observe that in Listing 1 only the line initializing the tracker is language-specific; data representation and program control are language-agnostic.

B. RISC-V Registers and Memory Viewer

We also used EasyTracker in our compiler courses, to develop a visualization showing the CPU registers and the memory represented as it is, hence a one-dimensional array of values.

Figure 7 shows how the tool looks. Again, thanks to the EasyTracker's expressiveness and its GBD-based implementation, it was easy to execute the program line by line and get register and memory values at each step. The visualization is here implemented using a splittable terminal showing side by side the source code (using `vim` in text mode) along with the memory image generated with the `dot` framework by the tool.

To inspect the program's state, registers and memory values in this case, we used the `get_registers_gdb` and `get_value_at_gdb` functions.

C. Recursive Calls Visualization

To help students grasp the control flow of the execution of a recursive function, we quickly implemented a dedicated visualization tool using EasyTracker. Figure 8 shows an example of the output of this tool. We can see a new node appearing in the tree at each recursive call. Red nodes are live calls. When a function exits, the tool changes the corresponding node color to gray. In the meantime, the return value is added to a back edge of the tree. This example focuses on understanding recursive calls; hence, each node displays the content of the

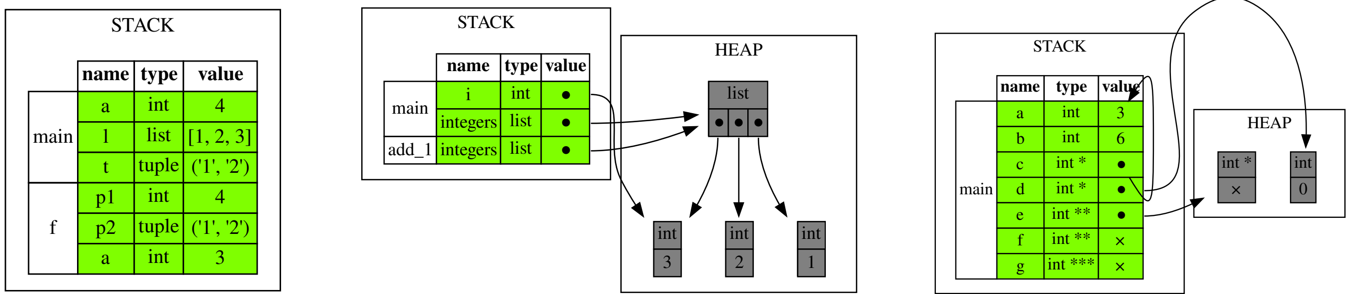

Fig. 6: From left to right: (a) Python stack diagram; (b) Python stack and heap diagrams; (c) C stack-and-heap diagram.

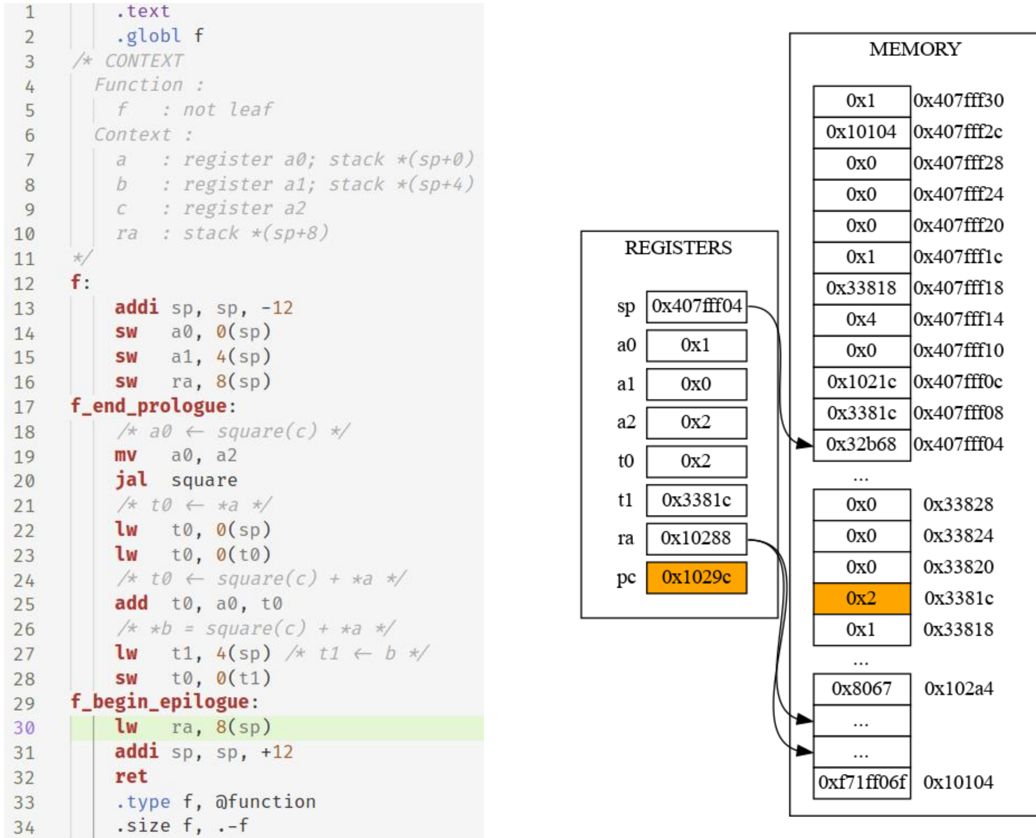

Fig. 7: RISC-V registers and memory viewer.

array at the time of the call even if it is a shared reference which content changes during the execution.

Listing 6 shows the corresponding control and visualization code. For the customization of the output, the user can specify the function and subset of variables to display, through `func_name` and `args_names` when calling the `control` function. Restricting the tracker to pause only at entry/exit of function `func_name` is performed by using **`tracker`**`.resume` along with **`tracker`**`.track_function(func_name)`. Also, compared to a tool processing afterwards a trace generated from a full step by step run of the program, the teacher can add interaction within its controller function, here querying for a `skip` value in order to choose a subset of the call trees to generate. As the visualization handles differently the events of entering (gathers the argument values we want to display and update the call tree) with the event of returning (label the recursive tree node as inactive), the reason of pausing is obtained through **`tracker`**`.pause_reason` on line 17.

D. A Game for Learning Debugging

In the context of another programming course using C, we used EasyTracker to build a game to learn debugging. Each level of the game consists of debugging a C program moving a character on a map. The goal for the player is to identify and fix bugs inside the level's program so that the character reaches the exit when the program is launched. Figure 9 shows what the game looks like. It is made of three parts: a gdb console

```python
#!/usr/bin/env python3

def m(x, elems):
    idx = len(elems) - 1 - x
    elems[idx] += 1
    if x == 0:
        return 1
    m1 = m(x - 1, elems)
    m2 = m(x - 1, elems)
    return m1 + m2

def main():
    r = 3
    elems = [0] * (r + 1)
    m(r, elems)
    print(elems)

if __name__ == "__main__":
    main()
```

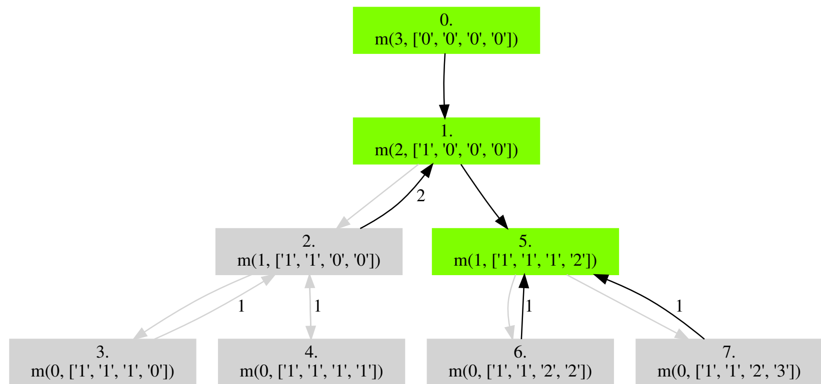

Fig. 8: Recursive call tree visualization.

on the left, a visual representation of the level's program on the center, and the source code of the level on the right.

Figure 9 also shows the control part for a simplified version of the game along with the C implementation of a level. The bug itself is a missing definition of `has_key` in the `check_key` function, hence when reaching the exit location for the level, the door stays closed as if the character did not pass over the key location. The control code lets the player modify the level source until the bug is resolved. Incrementally useful hints are given to the player in order to discover the issue. These hints are generated as the level is running by inspection of some useful program level variables. For the purpose of the example, in the given C code for the level, the movements of the character are simulated, though in practice they are non deterministic as the level is played.

This example implementation shows how the EasyTracker API keeps the program control and inspection accessible while the program is running, which is necessary when the visualization (here the debugging hints for instance) depends on the program control itself. This would not be possible with solutions based on a trace gathered from a full step-by-step run and processed afterwards. As shown in Fig. 9 the actual game controller uses the EasyTracker API to additionally manage the game layout, the interaction with the controller from the layout itself and updates of the debugging and source code views.

E. Interaction with Other Visualization and Trace Tools

One can also use the EasyTracker API to generate a trace or a partial trace for other visualization tools. For instance one can generate a PT trace for the example in Section III-C. As shown in Fig. 10, the PT front-end can then be used to walk the trace and visualize frames for the subset of variables chosen when generating the trace. Compared to generating a full trace, this allows to focus on interesting parts of the execution and reduce the trace by a factor of 10 in this example.

On the other hand, one can also use an existing trace format and navigate the trace with the EasyTracker API by implementing a dedicated tracker. It is possible for instance to add a PT tracker which reads a PT trace and implements the API on top of it. This enables the full power of control through the API on a pre-generated trace. Also it enables other languages supported by an external tracer and not supported by EasyTracker itself.

IV. Related Work

The survey of Sorva et al. (7) illustrates the importance of program and algorithm visualization (resp. PV & AV) for the computer science education community. In those works, the efforts mostly focus on evaluating the importance of enriched visualization of data structures. Still, they all share a common ground, where the taught notions are illustrated through the execution of a program or pseudo-code.

A. Comparison with Existing PV and AV Tools

Generally, for each visualization project, new instrumentation, tracing, interpretation, or rendering infrastructures are developed, partly due to the lack of decoupling between the program, its control, and its visualization as shown in Table I. For instance, while the quality of the outcome of JSAV (8) and VisuAlgo (9) is impressive, each algorithm has its own handwritten program (in JavaScript) that mixes the code that simulates the algorithm and the one for visualization and user interactions. These later tools do not provide any kind of decoupling. To decouple visualization with program/algorithm description, OGRE (10) uses a C++ interpreter. PlayVisualizerC/PVC.js (11) uses ANTLR for parsing C code, UNICOEN for building an AST, and UniTree to interpret it. The following tools chose to decouple the flow at the trace level. Vlsee (12) uses a Python transpiler to generate a `json` trace that describes the willing sequence of animations. Jeliot (13) uses DynamicJava interpreter to create an `XML` trace of the execution. SeeC (14) generates a trace using `clang` for code instrumentation in addition to a serialized AST for the post-processing. Eye (15) source emulates the execution of an AST (generated using `rply` from the C++ code) to generate a `json` trace enriched with a canonical graphic representation of some data structures (hash, stack, queues, trees – with highlighting of currently modified element). C Tutor (16) uses Valgrind (17) emulation to create a trace further processed by the visualization phase.

```
def control(prog, func_name, args_names):
 from types import SimpleNamespace as ns
 from easytracker.init_tracker import init_tracker
 from easytracker import PauseReasonType as prt
 from copy import deepcopy

 tracker = init_tracker("python" if prog.endswith(".py")
                        else "GDB")
 tracker.load_program(prog)

 current, calls, idx = None, [], 0
 tracker.track_function(func_name)
 tracker.start()

 while tracker.get_exit_code() is None:
  tracker.resume()
  reason = tracker.pause_reason
  if reason.type == prt.CALL:
   args = [tracker.get_variable_value(arg).value
           for arg in args_names]
   current = ns(args=args, uid=len(calls),
       child=[], parent=current, active=True,
       retval=None)
   if current.parent is not None:
    current.parent.child.append(current)
   calls.append(current)
  elif reason.type == prt.RETURN:
   current.retval = reason.args[2]
   current.active = False
   current = current.parent
  else:
   continue
  idx += 1
  visualize(f"rec-{idx:03}", calls[0], func_name,
            tracker.next_lineno, tracker.last_lineno,
            prog_name)

  tracker.terminate()

control("rec-program.py", "m", ["x", "y"])
```

```
def visualize(name, call, func_name):
 from subprocess import run
 from visualprimitives.source_image import (
  generate_source_image)
 with open(f'{name}.dot', "w") as f:
  f.write('digraph rec {\n')
  dump_call_tree(f, call, func_name)
  f.write('}\n')
 run(f'dot -Tsvg {name}.dot -o {name}.svg', shell=True)
 run(f'eog -w {name}.svg &', shell=True)
 generate_source_image(prog_name,f"{name}_src",
  next_lineno)

def dump_call_tree(dot_file, call, func_name):
 from easytracker import AbstractType as at
 def to_str(val):
  if val.abstract_type == at.PRIMITIVE:
   return str(val.content)
  elif val.abstract_type == at.LIST:
   return str([to_str(x) for x in val.content])
  elif val.abstract_type == at.REF:
   return to_str(val.content)
 args = ', '.join([to_str(arg) for arg in call.args])
 color = 'chartreuse' if call.active else 'lightgray'
 dot_file.write(
  f'{call.uid} '
  f'[color="{color}" style="filled"'
  f'shape="rect" margin="0" width="1"'
  f'label="{call.uid}.\\n{func_name}({args})"]\n')
 if call.parent:
  ecol = 'black' if call.active else 'lightgray'
  rcol = 'black' if call.parent.active else 'lightgray'
  rsty = 'style="invis"' if call.active else ''
  dot_file.write(
   f'{call.parent.uid} -> {call.uid} '
   f'[color="{ecol}"]\n')
  dot_file.write(
   f'{call.uid} -> {call.parent.uid} '
   f'[label="{call.retval}" '
   f'color="{rcol}" {rsty}]\n')
 for child in call.child:
  dump_call_tree(dot_file, child, func_name)
```

Listing 6: Recursive call tree EasyTracker `control` code and `visualize` code.

TABLE I: Properties of different classes of tools.

	Generality	Control	Inspection	Decoupling	Agnostic
AV/PV Tools	-	--	++	--	-
Instr. Tools	++	-	++	++	-
IDE	+	++	++	-	+
Debug MI	++	++	++	++	+
EasyTracker	++	+	+	++	++

TABLE II: EasyTracker versus debugger interfaces.

	Inferior languages	CtrlAPI	InspAPI	AbsModel
bdb (20)	Py	Py MidLevel	Py MidLevel	✘
pygdbmi (3)	C/Asm/Rust	Py LowLevel	Py LowLevel	✘
gdbwire (4)	C/Asm/Rust	C LowLevel	C LowLevel	✘
gdb/MI (2)	C/Asm/Rust	Text LowLevel	Text LowLevel	✘
EasyTracker	C/Asm/Rust/Py	Py HighLevel	Py HighLevel	✔

Unfortunately, decoupling at the trace level does not allow fine control over the recorded program state nor online visualization. Alternatively, using dynamic instrumentation tools such as Valgrind (17), DynamoRIO (18) or QEMU (19), would not provide control over the program execution nor language-agnostic property as shown in Table I.

B. Comparison with Debugger Machine Interfaces (MI)

There seems to be a lack of adoption of debugger's machine interfaces for PV applications. Our analysis, summarized in Table II, is that these interfaces propose an abstraction both for the control and inspection API, that are low-level and specific to either compiled or interpreted languages. While very close to debuggers MI in terms of properties, as shown in Table I, we chose to provide a language-agnostic and simple interface that non-compiler/debugger experts can use to develop PV and AV through a controller script. Just as debugger interfaces and as depicted in Fig. 1, one can decouple both program control and visualization instead of simply decoupling the visualization as trace-based tools described above. We actually use a *Program-Visualization-Controller* model and EasyTracker proposes the API for the Program-Controller part. Note that it is always possible to use it to generate a program trace. The trace can then be used with external visualization tools or in turn, as a source program with a trace interpreter.

C. Comparison with Debugger Front-ends and Development Environments

Most IDE such as Eclipse CDT (21), Emacs (22), vs-code (23), or Code::Blocks (24), but also some debugging

```python
def build_level(fname):
 from subprocess import run
 p = run(f'gcc -g -O0 -o {fname} {fname}.c', shell=True)
 return p.returncode

def ask_continue(name):
 return input(f'Please edit {name}.c then Enter or q: ') != "q"

def run_level_1():
 from easytracker.init_tracker import init_tracker
 from easytracker import PauseReasonType as prt

 name = 'level-1'
 exit_code = None
 tries, show_on_key, show_has_key = 0, False, False
 watch_vars = ["has_key", "key_x", "key_y", "player_x", "player_y"]

 while exit_code != 0:
  exit_code = build_level(name)
  if exit_code != 0:
   print('Program does not compile')
   if not ask_continue(name):
    break
   continue
  tracker = init_tracker("GDB")
  tracker.load_program(name)
  reason = tracker.start()
  for var in watch_vars:
   tracker.watch(var)
  check_bkp = tracker.break_before_func("check_key")
  cvars = {}
  while reason.type != prt.EXITED:
   if (show_on_key and reason.type == prt.BREAKPOINT and
       reason.args[0] == check_bkp and
       cvars['player_x'] == cvars['key_x'] and
       cvars['player_y'] == cvars['key_y']):
     print(f'The user moved over the key at {cvars["key_x"]}, {cvars["key_y"]}')
   elif reason.type == prt.WATCHPOINT:
    cvars[reason.args[1]] = reason.args[3]
    if show_has_key and reason.args[1] == "has_key":
     print(f'The "has_key" value changed from '
           f'{reason.args[2]} to {reason.args[3]}')
   reason = tracker.resume()
  exit_code = tracker.get_exit_code()
  if exit_code != 0:
   print(f'Bug still present!')
   if tries >= 1:
    print(f'Hint: "has_key" variable value is '
          f'{cvars["has_key"]}')
   if tries >= 2:
    print(f'Hint: you will be notified when '
          f'the player goes over the key')
    show_on_key = True
   if tries >= 3:
    print(f'Hint: you will be notified when '
          f'the "has_key" value changes')
    show_has_key = True
   if not ask_continue(name):
    break
   tries += 1
  else:
   print(f'Successfully fixed bug in {name}.c, '
         f'we get to the next level!')
  tracker.terminate()
 return exit_code

run_level_1()
```

```c
typedef enum { UP, DOWN, LEFT, RIGHT } orientation;

int player_x, player_y, exit_x, exit_y, key_x, key_y;
orientation player_o;
int has_key;

void msg(const char *msg) { puts(msg);}

void check_key() {
  if (player_x == key_x && player_y == key_y) {
    msg("You found a key !");
    // Player should set somewhere has_key = 1
    key_x = -1;
    key_y = -1;
  }
}

void verify_exit() {
  if (player_x != exit_x  player_y != exit_y) {
    msg("Did not find the door!");
    exit (EXIT_FAILURE);
  } else if (has_key != 1) {
    msg("Unable to open the door!");
    exit (EXIT_FAILURE);
  } else {msg("Completed Level!");}
}

void turn_left () {
  static orientation new_o[] = {LEFT, RIGHT, DOWN, UP};
  player_o = new_o[player_o];
}
void turn_right () {
  static orientation new_o[] = {RIGHT, LEFT, UP, DOWN};
  player_o = new_o[player_o];
}
void forward() {
  int inc_x[] = {0, 0, -1, 1};
  int inc_y[] = {-1, 1, 0, 0};
  player_x += inc_x[player_o];
  player_y += inc_y[player_o];
  check_key();
}

void init_level() {
  player_o = RIGHT; has_key = -1;
  player_x = 3; player_y = 2; key_x = 5; key_y = 1;
  exit_x = 7; exit_y = 0;
}

int main() {
  char simu[] = "fflffrff"; // Simulated for the example
  char *input = simu;

  init_level();
  while (*input != '\0') {
    switch (*input) {
      case 'f': forward(); break;
      case 'l': turn_left(); break;
      case 'r': turn_right(); break;
    }
    input++;
  }
  verify_exit();
  return 0;
}
```

Fig. 9: Screenshot of a game to learn debugging with EasyTracker; simplified controller; bogus C implementation of level 1.

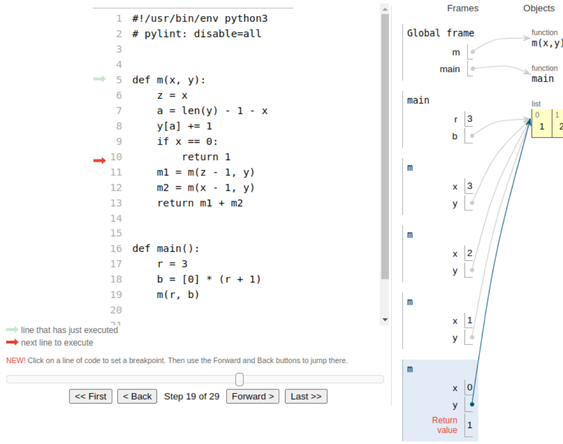

Fig. 10: A Python Tutor session from a EasyTracker trace.

TABLE III: How existing tools answer our needs.

	Inv	PyStkHeap	CStkHeap	RiscV	Rec	Agdb
ddd (31)	✘	✘	✘	✘	✘	✘
gdbgui (32), gdbfrontend (33)	More	✘	✘	✘	✘	✘
Thonny (34)	More	✘	✘	✘	✘	✘
VisuAlgo (9)	✔	✘	✘	✘	✘	✘
Python Tutor (16)	✘	More	More	✘	✘	✘
Recursion visu. e.g. (40)	✘	✘	✘	✘	Less	✘

frameworks such as KDbg (25), seer (26), Gede (27), or Insight (28) have very ergonomic human interface for controlling the program for debugging. Some of them such as PGDB (29) and PyDev (30) support parallel (mpi, multi-threaded) execution. Also a few debugger front-ends such as ddd (31), gdbgui (32) provide high level data structure visualization capabilities or even allow external plugins for extending visualizations as in gdb-frontend (33). Some IDE such as Python Thonny (34) simplify the beginner experience and may be used for visualization and learning. While using a visual debugger can sometimes help in teaching, they are usually considered as “not quite enough” (35), in particular, what they can show and how to control what/when to show is too restrictive. On the other hand, for most of our proposed applications, they display much more information than necessary as they manage the whole program state. Lastly, the front-end itself is the controller, while we aim at giving full control to the teacher through its control script, this is summarized in Table I. Examples of enhanced visualization not provided by visual debuggers include stepping on expressions (34, 36), showing stack and heap (37, 38), or adding role-based animation (39). As illustrated in Table III, our observation is that many specific teaching requirements are not covered by existing tools, which is precisely what motivated us to develop EasyTracker.

V. Conclusion and Future Work

Visual representations in computer science teaching are crucial to help students understand complex concepts.

We presented in this article EasyTracker, a library designed to help teachers develop their own specific program visualization tools. It provides an API granting developers the ability to control the execution of any program written either in Python or in any GDB-supported language, and to inspect its state at any time. EasyTracker also comes with several tools built on top of it that serve as both demonstrators and building blocks to be extended or modified to fit specific needs.

While we already use EasyTracker and its derived tools in our own courses, we also intend to work with high school teachers to help them cover their own needs.

We intend to keep on extending EasyTracker in the near future to make it usable for a wider range of languages. Any language supported by GDB is supported by EasyTracker for control commands, though the abstract representation of program state described in Section II-B2 has to be specialized. For instance, we aim at adding representations for the Rust basic types such as `Vector`, `String` and respective slices of these types. We are also studying the possibility of supporting the class of Java bytecode languages. Also, we have a preliminary implementation for a tracker supporting recorded execution through the RR (41) tool, allowing reverse execution or deterministic visualization of multi-threaded program races.

Currently, there are still some issues and limitations on the implementation side that we are working on. For instance:

- the python tracker has only been tested with the CPython implementation: it should be portable to others;
- the function tracking on C program in the presence of `setjmp`/`longjmp` is not supported yet;
- the tracking of multi-threaded Python programs has some race condition issues;
- there is some x86 architecture-specific code for C function tracking: it requires limited work to support new architectures;
- the control interface is synchronous: it is quite easy in Python to make it asynchronous, hence the choice. Though we may provide some API helpers to make it easier.

On the other hand, there are some limitations of uses which will not be tackled due to our design choices:

- the program control performance does not scale to a large number of control/introspection points compared to dedicated instrumentation tools, this is same limitation as a debugger;
- language specific objects not fitting the abstract model cannot be inspected with the base API, optionally, dedicated commands can be sent to the underlying tracker if portability accross languages is not a concern.

More specialized tools should be used for these usages.

In the current state, possible applications of EasyTracker that were not discussed and may be explored are:

- simultaneous control and visualization of multiple programs, for instance in client-server or distributed programming contexts;
- any kind of dynamic analysis (with limitation of scalability compared to specialized tools);
- inspection of hidden states in program components for unit testing;
- generation of partial and contextual traces for program equivalence testing.

VI. Data Availability Statement

Artifact allowing to reproduce images presented in this article are available (42).

Appendix

A. Abstract

This appendix describes the steps to reproduce the artifacts presented in this paper. All artifacts are examples usage of EasyTracker, hence, generated visualization figures or code listings. The artifacts generation is done through a set of scripts contained in the published artifacts archive. The top level script sets up the python environment and executes for each figure a dedicated script which reproduces it. The listing of the controller code and visualization is available in the artifact archive or in the example tools of the EasyTracker repository.

B. Description

1) How Delivered: The artifacts are delivered through the DOI 10.5281/zenodo.10428215 containing the artifacts archive, the archive of EasyTracker and the archive of the Docker image.

2) Hardware Dependencies: The running system must be a Linux x86_64 distribution. Preferably an Ubuntu 22.04 system on which the tools have been tested. The docker image provided is itself built on top of an Ubuntu 22.04 system.

3) Software Dependencies: In order to improve reproducibility, a Docker image is provided, in this case the only dependency for reproducing the artifacts is docker.

Otherwise, for reproduction on a native distribution, the minimal dependencies are:

- python, version 3.9 or higher
- gdb, version 11.2 or higher
- a working compiler toolchain
- graphviz (for graph generation)

The minimal versions of python and gdb are mandatory, otherwise unexpected python exceptions will raise. To display generated figures, one may install a viewer such as `eog`.

C. Installation

First download the artifacts archive, EasyTracker archive and docker image from the artifacts repository at https://doi.org/10.5281/zenodo.10428215.

Then extract the sources archives:

```
tar xzf easytracker-artifacts-cgo-2024-v1.2.0.tar.gz
cd easytracker-artifacts-cgo-2024
tar xzf ../easytracker-archive-dfe8aa888f.tar.gz
```

And load the docker image with:

```
docker load -i docker-image-easytracker-1.2.0.tar
```

D. Artifacts Generation

In order to generate all the figures and listing, execute:

```
./in-docker.sh ./run-all.sh
```

One may also proceed step by step for each figure generation, refer to the `README.md` file for more details.

E. Expected Figures and Listings

After the generation step, the following images are available for the different figures:

- Figure 1: array invariant visualization
  - source: figure-1/source21.jpg
  - array: figure-1/array21.svg
- Figure 6: stack anf heap visualization
  - 6a: figure-6/figure-6a/008-stack_heap.svg
  - 6b: figure-6/figure-6b/007-stack_heap.svg
  - 6c: figure-6/figure-6c/014-stack_heap.svg
- Figure 7: RiscV registers and memory viewer
  - source: figure-7/riscv_src.jpg
  - memory and registers: figure-7/riscv_viewer.png
- Figure 8: recursive function call tree visualization
  - source: figure-8/rec-013_src.jpg
  - call tree graph: figure-8/rec-013.svg
- Figure 9: debugging game
  - see below for listings
  - game snapshot: figure-9/game-display.png
- Figure 10: python tutor visualization
  - generated HTML: figure-10/ptv3/demo.html
  - open the HTML file with a browser and use the ”Forward” button until step 19 to see the figure

Listings given in the paper are available at:

- Listing 6: recursive tree visualization
  - control and display in paper-examples/recvis/recvis.py
- Listings of Figure 9: debugging game
  - controller sources in paper-examples/interactive/game.py
  - code to debug in paper-examples/interactive/level-1.c

F. Experiment Customization

One may run independently the scripts for generating the figures and optionally modify the controller codes executed from the scripts, refer to the `README.md` file for details.

Actually EasyTracker can be installed independently of this artifacts generation process and used on its up-to-date versions.

Refer to the home page and sources of EasyTracker at https://corse.gitlabpages.inria.fr/easytracker and https://gitlab.inria.fr/CORSE/easytracker.git.

G. Notes

Two figures from the paper are just provided as is:

- Figure 7, the RiscV memory/registers view: this figure was contributed by a teacher using a RiscV toolchain which we could not rebuild
- Figure 9, the snapshot of the Debugging Game: this figure was provided by the game writers and we could not reproduce it without providing the full game sources which is not in a publishable state. We provide, though, in the paper a very simplified program giving an idea of the game play, and this artifact is provided.

REFERENCES

[1] Philip Guo. *Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia*, page 1235–1251. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450386357. URL https://doi.org/10.1145/3472749.3474819.

[2] R. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. 12th Media Services, 2018. ISBN 9781680921434.

[3] pygdbmi, accessed August, 2023. URL https://github.com/cs01/pygdbmi.

[4] gdbwire, accessed August, 2023. URL https://github.com/brasko/gdbwire.

[5] Call stack as diagram, 2010 (accessed January, 2022). URL https://notionalmachines.github.io/nms/CallStackAsDiagram.html.

[6] Paul E. Dickson and Toby Dragon. A memory diagram for all seasons. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 150–156, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382144. doi: 10.1145/3430665.3456317. URL https://doi.org/10.1145/3430665.3456317.

[7] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.*, 13(4), nov 2013. doi: 10.1145/2490822. URL https://doi.org/10.1145/2490822.

[8] Ville Karavirta and Clifford A. Shaffer. Creating engaging online learning material with the jsav javascript algorithm visualization library. *IEEE Transactions on Learning Technologies*, 9(2):171–183, 2016. doi: 10.1109/TLT.2015.2490673.

[9] Steven Halim. Visualgo–visualising data structures and algorithms through animation. volume 9, pages 243–245, 2015.

[10] Iain Milne and Glenn Rowe. Ogre: Three-dimensional program visualization for novice programmers. *Education and Information Technologies*, 9:219–237, 2004.

[11] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. Pvc.js: visualizing c programs on web browsers for novices. *Heliyon*, 6(4):e03806, 2020.

[12] Teemu Sirkiä. Exploring expression-level program visualization in cs1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, page 153–157, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, 2004.

[14] Matthew Heinsen Egan and Chris McDonald. Program visualization and explanation for novice c programmers. In *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*, pages 51–57, 2014.

[15] Aman Bansal, Preey Shah, and Sahil Shah. Eye: Program visualizer for cs2, 2021.

[16] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 579–584, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445368. URL https://doi.org/10.1145/2445196.2445368.

[17] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL https://doi.org/10.1145/1273442.1250746.

[18] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275, 2003. doi: 10.1109/CGO.2003.1191551.

[19] QEMU: the FAST! processor emulator. https://www.qemu.org.

[20] Python documentation. python bdb – debugger framework, accessed August, 2023. URL https://docs.python.org/3/library/bdb.html.

[21] Eclipse cdt, accessed August, 2023. URL https://github.com/eclipse-cdt.

[22] emacs, accessed August, 2023. URL https://www.gnu.org/software/emacs/.

[23] vscode, accessed August, 2023. URL https://code.visualstudio.com/.

[24] Code::blocks, accessed August, 2023. URL https://www.codeblocks.org/.

[25] kdbg, accessed August, 2023. URL https://www.kdbg.org/.

[26] seer, accessed August, 2023. URL https://github.com/epasveer/seer.

[27] gede, accessed August, 2023. URL https://gede.dexar.se/.

[28] insight, accessed August, 2023. URL https://sourceware.org/insight/.

[29] Pgdb, accessed August, 2023. URL https://github.com/ndryden/PGDB.

[30] pydev, accessed August, 2023. URL https://www.pydev.org/.

[31] ddd, accessed August, 2023. URL https://www.gnu.org/software/ddd.

[32] gdbgui, accessed August, 2023. URL https://www.gdbgui.com/.

[33] gdb-frontend, accessed August, 2023. URL https://github.com/rohanrhu/gdb-frontend.

[34] Aivar Annamaa. Introducing thonny, a python ide for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, page 117–121, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340205. doi: 10.1145/2828959.2828969. URL

https://doi.org/10.1145/2828959.2828969.

[35] Juha Sorva. *Visual program simulation in introductory programming education*. PhD thesis, 2012. URL https://api.semanticscholar.org/CorpusID:196101569.

[36] Juha Sorva and Teemu Sirkiä. Uuhistle: A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, page 49–54, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450305204. doi: 10.1145/1930464.1930471. URL https://doi.org/10.1145/1930464.1930471.

[37] Paul Gries, Vlad Mnih, J. Taylor, G M Wilson, and Lee Zamparo. Memview: a pedagogically-motivated visual debugger. *Proceedings Frontiers in Education 35th Annual Conference*, pages S1J–11, 2005. URL https://api.semanticscholar.org/CorpusID:15206550.

[38] M.P. Bruce-Lockhart and Theodore S. Norvell. Developing mental models of computer programming interactively via the web. *2007 37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports*, pages S3H–3–S3H–8, 2007. URL https://api.semanticscholar.org/CorpusID:35401513.

[39] Jorma Sajaniemi and Marja Kuittinen. Program animation based on the roles of variables. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, page 7–ff, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136420. doi: 10.1145/774833.774835. URL https://doi.org/10.1145/774833.774835.

[40] Recursion visualizer, accessed August, 2023. URL https://www.recursionvisualizer.com/.

[41] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability: Extended technical report, 2017.

[42] Christophe Guillon. Artifacts of the CGO 2024 Paper: EasyTracker: A Python Library for Controlling and Inspecting Program Execution, December 2023. URL https://doi.org/10.5281/zenodo.10428215.