Preface: *There are 2 primary additions contained in this second draft: an expanded introduction which now includes visualizations to explain how I am reasoning about arguments and the newly added section on guile. In the process of adding these things, I have gathered some doubts about how I am approaching this paper.*

The biggest concern I have is the treatment of intermediate representations as black boxes. I restrict myself to examining the source code as input and whatever representation the language considers to be the "final" output - for Python and Guile this is bytecode, but for C this will be machine code. This seems useful to contain the scope of the paper, as otherwise the analysis and description parts could get out of hand (if they are not already). However, I am worried that ignoring the intermediate representations prevents me from attaining certain insights. The purpose of examining the output of compilers is to understand how the computer is processing the information given to it by the human and to inform the implementation of the framework. The end result gives a concrete and literal understanding of what actions the computer decides to take based on the human's input, but it doesn't tell me how the computer arrived at those conclusions or why it made the decisions it did. What happened during the Guile section is that I looked at intermediate representations in order to inform my description of the language, but I did not directly discuss these representations in the paper. This might be the best compromise, and it is how I will proceed for the time being.

I am also beginning to doubt the idea of having a separate section for the "simple function and call". The problem is that it undercuts the positional value section because the simple function and call is so similar to the positional value examples. However, I am still partial towards including this section. First, it avoids priveliging any particular feature over the others. While it is true that all of the languages in this paper use positional values as the default (with Python fitting this idea less well than the others), this is mostly an historical accident and not due to the fundamental nature of function arguments. For example, the Modula-3 language definition (https://doi.org/10.1145/142137.142141) states "The list of bindings is rewritten to fit the signature of `P's` *type as follows: First, each positional binding* `actual` *is converted and added to the list of keyword bindings by supplying the name of the* `i'th` *formal parameter, where* `actual` *is the* `i'th` *binding in* `Bindings`*." Also, there are sometimes positional-specific aspects to discuss in the positional section which are not relevant to the simple function and call; for example, in Python it is considered simpler for a function to accept either positional or named values than it is to accept positional values only (this is quite abnormal relative to the other languages in this paper). Second, it gives a clear starting point for someone who is looking at the paper and only cares about some of the features. The section title "Simple Function and Call" clearly indicates to the reader that this is an expectation-setting section which should be read regardless of the reader's particular interest. In contrast, it is less intuitive to read the positional values section if one is mainly concerned with, for example, default values.*

Additionally, I am uncertain what subject of study this paper actually falls into. Looking at the paper by word count, much of it is focused on the way that programming languages operate and accept input - so, computer science right? Except that my primary concern is not about the computer itself, but the way that humans use source code to communicate, both with the computer and with each other. So is this information science because I'm concerned with the transmission of information? Is it sociological because I'm concerned with human-to-human interaction, albeit indirect and asynchronous interaction? Or is it computer science that (should) draw on other fields in order to enhance the validity of the work? I'm not sure. I'm also uncertain how useful it is to precisely define what the "subject of study" is, or if it is even realistic to expect that any given work will fit neatly into exactly one subject of study.

Finally, I am growing increasingly skeptical that LaTeX is the best tool for publishing, particularly when when compared to web technologies (HTML, CSS, JavaScript). LaTeX both creates extra work for me and fails to provide as many features as the web. For example, the current draft includes many code boxes with code that spills out of the box. Many of

the code samples which fit the box have poor formatting in order to make them fit. In some cases it is my example code which can be modified, but in other cases it is code samples which come from third parties; modifying these would be dishonest. Fixing this problem wastes my time as the author and decreases readability for the audience. Web technologies solve this problem elegantly by providing a scroll bar for text that spills over. Web technologies also provide other useful dynamic features. For example, if the text size is too small for a particular reader they can zoom in on a PDF but this will force the reader to constantly scroll around the page which is particularly inconvenient in the two-column format (I know this from personal experience). Modern web browsers implement zooming by changing the font size and automatically updating the page layout based on style rules. Web technologies also allow me to use more powerful presentation tools in my paper. For example, a code sample which is associated with a series of paragraphs can be attached to that text in the margin and scroll along with the reader so it is always accessible (by using the "sticky" attribute). In places where I reference a previous code sample - for example when I compare the bytecode of a specific feature to the bytecode for the simple function/call - I can implement a pop-up box that appears when the user hovers over the text for ease of reference. Of course, this is still a paper not a website. The paper will continue to be published as a single portable self-contained document. The fact that the document opens in Firefox instead of Evince will not change this important attribute.

On the topic of more powerful publishing features and pop-up boxes, I feel the need to address my citations. I understand that the citations in the current draft are extremely cumbersome to read past. However, I also feel that providing precise location information is important. The purpose of citations are not simply to justify my claims, they are meant to help a reader who wants to explore a particular aspect of the paper, or who wants to examine a tangentially related topic. I have frequently been frustrated when reading other papers that they make claims about how programs function but provide no information about how they justify this claim - whether through source code, documentation, or simply running the program. I do not want any of my readers to have a similar frustration. With the move to web technologies, I can have WikiPedia-style citations which are simply a number that pops out a box containing detailed information when hovered. This will help me achieve my goal of providide precise information relevant to the specific citaiton index without inflating my citation count artificially (for example, by keeping the Guile repository as a single citation but continuing to provide file and line numbers through the pop out box).

The next draft of this paper will primarily focus on converting the format to web technologies with minimal changes to content. However, I also want to explore what the analysis will look like. This seems complicated because I want to discuss each feature in isolation but I also want to discuss how they interact with each other. There are far too many features to describe each possible pairing, and even if I did this would still not be complete. For example, named values, default values, and caller destructuring could interact in ways that are not fully describable in isolated pairings. I will not really know what this part will look like until I finish describing all of the languages but I want to start trying things out now to get the brain juices flowing. =)

Comparative Function Arguments

Skyler Ferris

1 Introduction

Functions are one of the core abstractions that programmers use. These functions - and their authors - have to communicate with different calling sites - and their authors - in different contexts. This communication is performed through the use of arguments. However, the term "argument" is used in different ways depending on the speaker, listener, and context. For example, people who are working on a new function written in the C programming language might use the word "arguments" to refer to the set of names that appear between the parantheses in the function signature. Those same people who are later debugging code which includes a call to that function might use the word "arguments" to refer to the values which are given to the function. This informal use of the word is sufficient for day-to-day use, but it starts to break down when examined more closely.

Consider that the Rust programming lanugage allows programmers to specify arguments using one of two mechanisms. In the common case, programmers can use a plain name and type:

```
fn jump_plain(starting: &Point) -> Point {
  Point {
    x: starting.x,
    y: starting.y * 2
  }
}
```

It would not be controvertial to claim that this function contains one argument. This argument happens to be associated with a local variable named `starting` which is guaranteed to contain data of type `Point`.

When it is useful, programmers can also specify an argument by a pattern [1] and type:

```
fn jump_pattern(Point {
                  x: h, // horizontal
                  y: v  // vertical
                }: &Point)
  -> Point {

  Point {
    x: *h,
    y: *v * 2
  }
}
```

Does this version contain one argument or two? We could say that it contains 2 arguments, `h` and `v`. In the context of the function definition this would make sense. However, at the calling site both appear to accept only a single argument:

```
jump_plain  (&uut);
jump_pattern(&uut);
```

Consider also the conventions in shell programming languages. Shell functions might include "options" which are arguments that may or may not be related to the argument that follows it. For example, consider the following function:

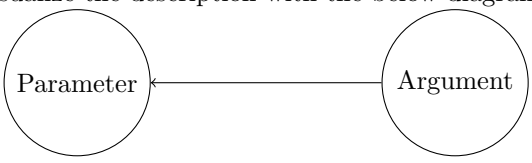
[1] Technically the "plain name" is also a pattern, just a very simple one.

```
function main {
  PRINT_VERSION=false
  declare -a IGNORE_LIST

  GETOPT_OUTPUT=$(\
    getopt -o     'vi:' \
           --long 'version,ignore:' \
           --     "$@")

  eval set -- "$GETOPT_OUTPUT"
  unset GETOPT_OUTPUT

  while true; do
    case "$1" in
      '-v'|'--version')
        PRINT_VERSION=true
        shift
        continue
      ;;
      '-i'|'--ignore')
        IGNORE_LIST+=("$2")
        shift 2
        continue
      ;;
      '--')
        shift
        break
      ;;
    esac
  done

  if [ "$PRINT_VERSION" = true ]; then
    echo "Version 0.1"
  else
    echo "${IGNORE_LIST[*]}"
  fi
}
```

Some of the ways this function could be called include (1) `main --ignore=foo`, (2) `main --ignore foo`, (3) `main -ifoo`, or (4) `main -i foo`. All of these calls are semantically synonymous. However, in 1 and 3, the bash interpreter initially sees only a single argument while in 2 and 4 it sees 2 separate arguments. In either case, the call to `getopt` reorganizes the arguments so that the name (either "--ignore" or "-i") is separate from the value ("foo"). This makes the "number of arguments" an ambiguous value.

1.1 What is an argument?

So, how can we construct a definition of "function argument" which does not suffer from the above deficiencies? The definition provided by the C standard is a good place to start: it precisely defines 2 relevant terms, parameter and argument [7, Section 3]. It defines a parameter as an "object [2] declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition". It defines an argument as an "expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation". Additionally, the standard defines a relationship between arguments and parameters: "In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument" [7, Section 6.5.2.2 paragraph 4]. We could visualize the description with the below diagram.



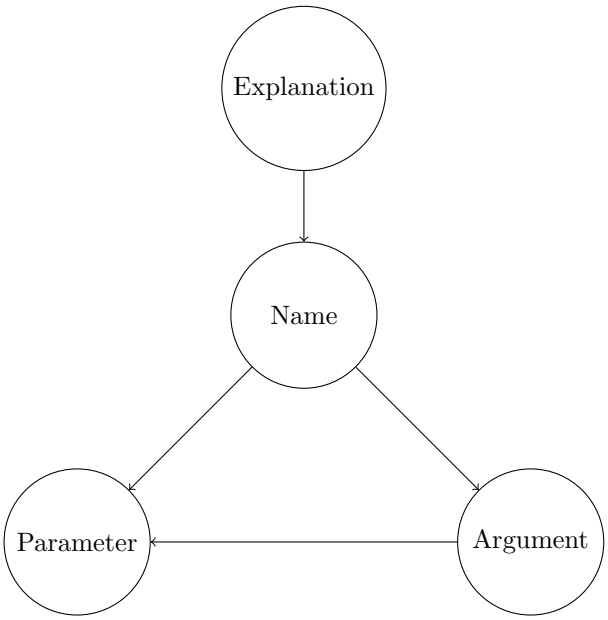
Here the circles represent concepts within the C model of computation - the local variable declared by the parameter and the value given as an argument. The line between them represents their relationship: the expectation that the local variables will be initialized with the values.

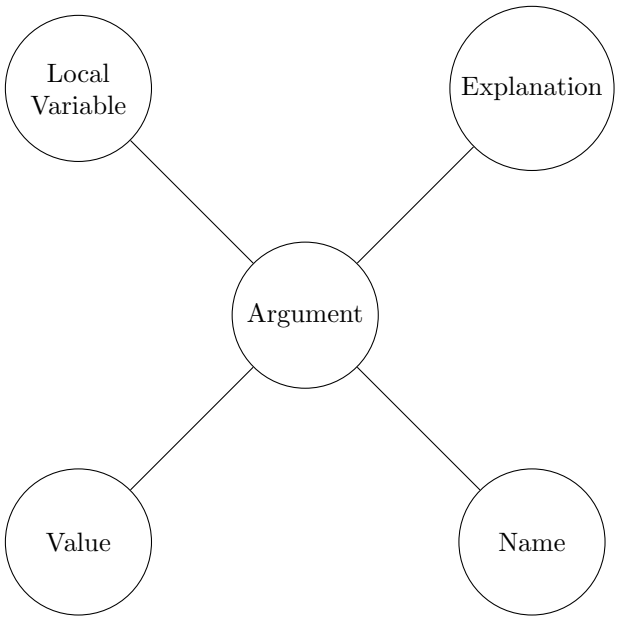
While this visualization fully encompasses the model described by the C standard, it does not fully encompass the mental model used by some programmers. For example, consider this C function:

```c
/*
 * Calculate the division of two numbers.
 *
 * lhs: The dividend
 * rhs: The divisor
 */
float divide(float lhs, float rhs) {
  return lhs / rhs;
}
```

The comment above the function reveals 2 elements in the author's mental model which are absent from the visualization: an explanation and a name. The explanation, such as the text "the dividend", communicates the intention of an argument's use from one person (an author) to another (a caller). The comment also clarifies that the names "lhs" and "rhs" are not simply the textual representation of local variables: they refer to the argument itself, even in contexts where the local variable is not directly relevant. In order to accommodate this, we need to update the visualization with "name" and "explanation" nodes. The explanation node is connected directly to the name a established by the comment. It is indirectly connected to both the parameter and the value through the name.

[2] Use of the term "object" here refers to any region of memory, not the concept of an object popularized by object-oriented programming.





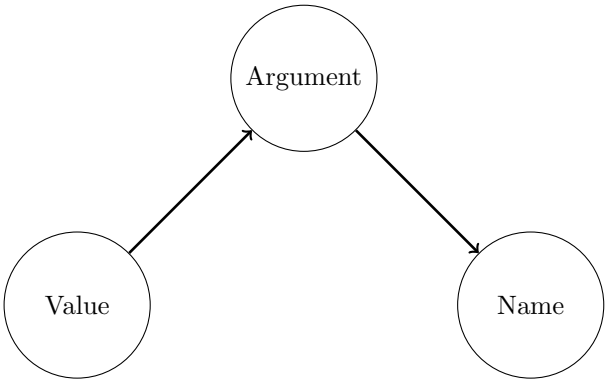
This "hub and spoke model" allows us to attach relavant concepts to the argument as a whole. Each processing tool can search for the concepts it needs by checking which things are attached to the argument without worrying about which tools happen to be processing the argument. When we want to consider a specific tool, we can modify the graph by trimming unused nodes and adding arrows that are relevant to the tool. For example, consider a tool which assesses the complexity of lines of code in a program. It might encounter a line like this:

```
divide(4, c=(a + (b * 2) / 3));
```

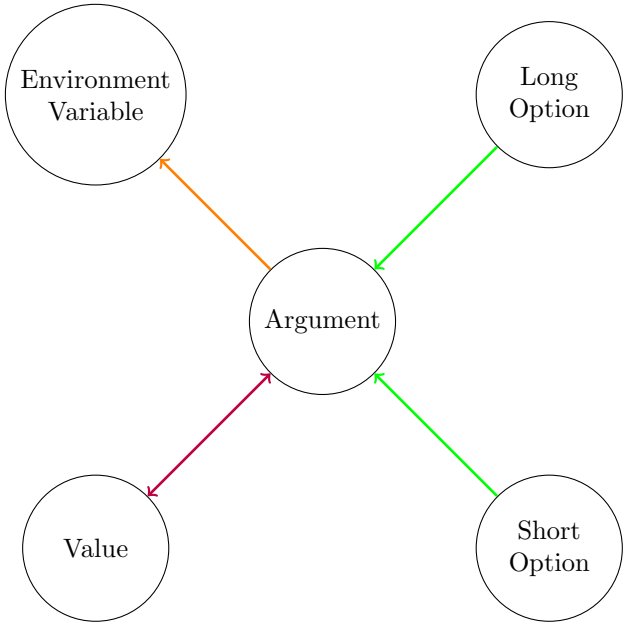
This line would probably be rated as complex due to the use of assignment as an expression, and possibly due to the number of mathematical operations in the expression. The tool could simply reprt the line number but in complicated programs it might also be useful to specify which argument is the source of the problem. That is, instead of simply saying "on line number x" it could say "in the value given to the rhs argument in the call to divide on line number x". In order to determine the argument name it would first look for the argument that it associated with the source of the complexity. Then it looks for the name associated with the argument in order to

Unfortunately, this method of building the visualization will quickly become unwieldy. As we add more tools requiring more concepts the graph will grow in both size and complexity, as each tool might add new associations between nodes leading to a tangle of lines. It is more effective to consider the "argument" to be an abstract concept which connects the concrete things together; in order to accommodate this, I will use the term "value" to refer to the concrete thing that C refers to as an "argument":

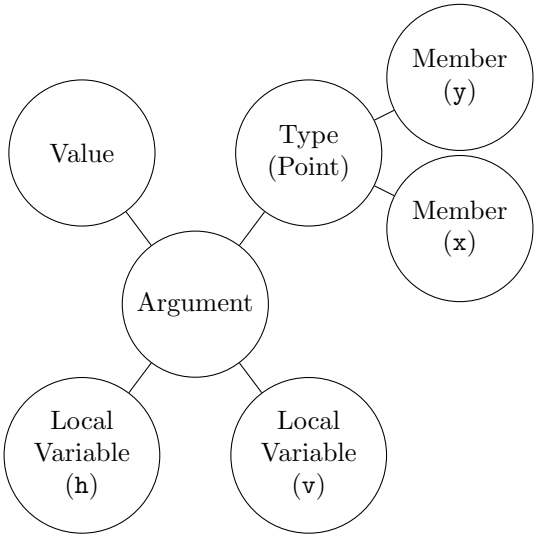
understand how to communicate effectively with the human:

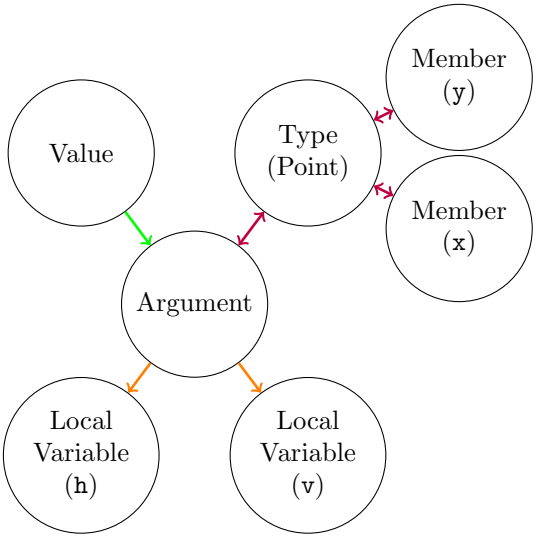


This method solves the problem of connections between nodes growing out of control. As another example, in the bash program we would want to include the concept of option names as well as the the variable which ultimately receives the value. Additionally, Bash exposes some of the processing which is implicitly performed by the compiler in C. This makes it a useful example to demonstrate how actions typically taken by the compiler/interpreter can be mapped onto the visualization. I will focus on the `case` statement which determines which given values should be associated with which environment variables. First, the `case` statement examines the short or long option to determine which argument is being provided. This is represented by the green arrows in the below diagram, moving from the option to the argument. Next, the logic inside the relevant case determines which value is associated with the argument. The "version" argument determines the value by the mere presence of the name while the "ignore" argument determines the value by retrieving a value provided by the caller. Both methods are represented by the double-ended purple arrow, representing that the interpreter retrieves the value then returns to the argument for further processing. Finally, the value is placed into an environment variable. This is represented by the orange arrow, moving from the argument to the environment variable and "carrying" the value with it.



The Rust program produces a more complex graph due to the presence of the pattern. The compiler processes the pattern by looking at the data provided in a structured manner; it knows that a specific local variable should be initialized with a subset of the bits provided by the value. These subsets are not associated with the argument. Instad, they are associated with a data type which is associated with the argument. Where the Bash graph was relatively simple, with all nodes other than the Argument node containing exactly one connection to the Argument node, the Rust graph contains a subgraph attached to the Argument node.





When the Rust compiler encounters a call to this function it first retrieves the value from the calling site. This is represented by the green arrow in the below diagram, moving from the value to the argument. Next, it looks at the members of the Point structure in order to determine which subsets of bytes are semantically relevant. This is shown in the double-ended purple arrows which represent the compiler performing this lookup and returning to the argument with this information in hand. Finally, it associates the relevant subsets with the relevant local variables as shown by the orange arrows moving from the argument to the local variables.

Visualizing arguments in this manner leads me to define an argument as an abstract concept which emerges from the way that processors (both computer and human) associate concrete things. This paper will explore the implications of this definiton by describing the common and unique features associated with function arguments across several languages. Additionally, it will present a mechanism implemented in GNU Guile which frees programmers to create arbitrary associations when declaring arguments. This will generalize the features provided by various languages, allowing programmers to use the set of features that make sense given the context they are working in without needing to change the language that they are programming in.

1.2 Practical Applications

While the theoretical problems with the term "argument" compel me to investigate this phenomenon, some people might ask whether this actually makes a difference to people who are working on computers to achieve practical ends. A good question! As you might have guessed from the presence of this paragraph, this does have implications for practical work. My initial intuition that this is practically useful comes from the ubiquity of function calls in all programs across different programming paradigms. Even in object-oriented programming, a significant

portion of lines of code consist of function calls. Any improvement to the function call interface will have an outsized effect on developers due to the sheer volume of function calls we have to deal with. Additionally, my intuition is backed by observing the industry, including which technologies gain and maintain popularity as well as the way people interact with them.

Consider the way that bash commonly processes arguments. It is often the case that a caller can supply different values for the same argument in the same call, with the values that are syntactically later overriding the was that are syntactically earlier. When working on a command line this can be useful. If the caller wants to re-run a command but change one argument they do not need to navigate through the invocation and edit the correct spot; instead, they can add a new value onto the end of the call to get the desired behavior. In an interactive setting where the mental flow of the caller is of the utmost importance this feature is useful. However, it would be considered bad form to do this inside of a shell script. While Bash technically allows this to happen, as it is a prompt-first language, other popular languages such as Python which is a script-first language disallow this behavior. There is no "right answer" as to whether or not the language should allow this: it is a question of context. This paper will define semnatics like this as metadata associated with arguments: metadata which can be interpreted differently or changed completely by tools specialized for specific contexts (for example, an interpreter could behave differently if it is given a script on the command-line compared to when it is running an interactive session).

The contextual differences extend to social settings as well. For example, as we will see in the section on Python it is possible for an author to specify that some values cannot be given by name (by default, callers may choose whether to pass values by position or name freely). One of the justifications given for this is that it simply does not make sense to use names for some functions in any context. The example given is a casting funciton with the signature `as_my_type(x)`. Here, they argue, it will never make sense to use the name `x` because it does not provide any meaning. However, a teacher in a Programming 101 class might decide that they want to name all values in their code examples and assignments, so that the code is clearer for people who are learning to program for the first time. While they could omit the name for some simple functions such as `as_my_type` this adds conceptual complication to the material: now students must switch between processing positional values and named values on a case-by-case basis, or else completely abandon named values. By specifying these restrictions through accessible metadata, the teacher could simply remove this restriction for the purpose of their class without disturbing the upstream project.

1.3 Reproducibility and Transparency

The source code used in example snippets can be found in the online git repository located at http://git.sr.ht/~skyvine/comparative-function-arguments. The repository includes files appropriate for use with GNU Guix to reproduce the software environment I used while creating this paper. It pins to a specific revision of the main Guix channel so that updates do not interfere with reproducibility. The Makefile within the "examples" directory launches a pure shell[3] for the user so that environmental factors do not interfere with reproducibility.

The git repository also contains the bibtex file used to generate references in this paper. This source file contains additional information which is not present in the output, such as the SHA256 of referenced tarballs and the commit hash of source repositories. [4]

1.4 Clarifying Terminology

As mentioned above, this paper defines the term "argument" to refer to a set of associations between concrete code objects, such as values and variables.

[3] Here, "pure" means that enviornment conditions such as variables are unset so that the shell does not use any artifacts from the base operating system by mistake.

[4] I intend to incorporate this information into the references in the final version of this paper, but I have not used Latex before so I will need to learn how to do that.

Some languages provide similar features under different names and sometimes those names do not align with the vocabulary used by this paper. For example, different communities use different terms to refer to the values that shell functions interpret as implying a value. A shell function might recognize a value "--verbose" to mean that a variable named "verbose" should have the value "true". I have heard this mechanism variously referred to as a flag, a switch, or an option. This paper refers to this mechanism as an "implicit value" because the caller uses a name associated with the argument to imply a value which does not appear explicitly. A complete least of terms which describe a feature provided by more than one language follows:

Positional Value *(common name: N/A)*: These values are associated with arguments based on the index of the value in the list of all positional values.

Default Value *(common name: Optional Argument)*: A value that an author associates with an argument for use when the caller declines to supply a value.

Named Value *(common name: Keyword Argument, Option Argument* [5]*)*: These values are associated with an argument based on a name which the caller attaches to the value. The argument may be associated with several **synonymous** names.

Implicit Value *(common names: Switch, Flag, Option)*: An argument has an implicit value if the caller can specify a name associated with the argument but omit the value. For example, many CLI tools have a "--verbose" flag which increases the amount of output produced. An argument with an implicit value may also have **antonyms**, which invert the semantics of the primary name. For example, some CLI tools also have a "--quiet" option which disables verbose output and possible disables some non-verbose output.

Typed Value *(common name: N/A)*: Some languages allow or require associating a type with an argument, in which case the value provided by the caller must be compatible with that type.

Destructuring *(common names: Pattern, Unpacking)*: Some languages provide a mechanism to break a composite structure into its component parts. In some cases this functionality is specified by the author, in others by the caller. In both cases the caller provides a value of a specific type and different parts of that value (for example, different elements in a list or different members of a struct) are bound to different local variables in the body of the function. The difference is whether the author decides how the value is destructured (as in the case of Rust's patterns) or the caller does (as in the case of Python's argument unpacking). It is theoretically possible for a language to provide both of these variants simultaneously.

Variadic Function *(common name: N/A)*: A function which can accept an arbitrary number of values. Variadic functions are distinct from arguments with default values because with default values the author controls the range of acceptable argument counts and each value provided by the caller is bound to a different local variable. With variadic functions, the caller can pass in an arbitrarily large number of arguments (physical limitations notwithstanding) and the values are typically aggregated into a container; the variable bound to this container is the **variadic variable**. **Variadic values** are values which are associated with this feature. For example, a variadic function may require one positional value at the beginning of the value list. The first value is not a variadic value but all other values are.

1.5 Structure of Paper

The remainder of this paper will be divided into 3 parts: Description, Analysis, and Implementation. The purpose of these parts is to first understand what the current state of the craft is, then assess the advantages and disadvantages of different approaches, and finally create a framework that allows programmers to make context-aware decisions in an interoperable manner. The description section simply explains the way that a sample of programming languages process function arguments. The analysis section compares these approaches to each other and comments

[5] The term "option argument" is not to be confused with the term "optional argument". The former is used by shell users to refer to an argument that come after an option (such as "--ignore" "foo" in the previous example) while an optional argument is a term used by scripters and system programmers to refer to an argument associated with a default value.

on the appropriateness of each approach for different contexts. The implementation section describes how the framework was developed and how it operates in practice.

Part I
Description

2 Overview

This part is dedicated to describing the features of various languages that are sampled by this paper, using the terms defined by this paper. Each of these sections will start with a summary of the features provided by the language. Next, it will explain the background knowledge necessary in order to understand the feature descriptions. This explanation will include an example of a "simple function and call" which serves two purposes. First, it provides the reader with a concrete example of what will be described in the context of this language. For example, the description in Python will focus on the behavior of the interpreter and the bytecode emitted by the compiler. Second, it provides a point of comparison when describing features. In Python, the description of destructuring primarily describes the `CALL_FUNCTION_EX` instruction, with some description of list-building instructions towards the end, rather than describing every single instruction that is emitted. This keeps each section shorter because they do not have to explain the baseline they are being compared to; the sections are not self-contained, but they all have exactly one dependency and the dependency plus the section are self-contained. This makes it easier for a reader to focus only on the features that are relevant to them, if they so choose. Finally, each of the features provided by the language will be described.

Each feature description will start either with a statement that the feature is not provided or a statement of how the feature is provided. For example, in Python positional parameters are the "default behavior". Tutorials (including the official tutorial contained in the repository) commonly introduce functions by using positional parameters and introduce "keyword arguments" as a separate feature at a later point. Therefore, the Python section about positional values starts with "provided by default" while the section about named values starts with "provided

through keyword arguments". This helps the reader understand the feature's relationship to the language and clarifies what terminology will be useful if they are reading language documentation or performing an internet search. After this statement, any distinctive qualities of the feature as provided by the language will be noted.

Finally, there will be 3 subsections: Syntax, Implementation, and Historical Record. All 3 of these sections inform the implementation of the mechanism and perform a service for the reader [6].

The syntax section explains what the source code looks like when the feature is used. This helps readers who are unfamiliar with the language in question understand the code snippets in the following subsections.

The implementation section explains the language behavior which causes the feature to be provided. This helps clarify the "concrete things" that the argument is associating.

The historical record section discusses, where possible, the motivation for the feature and lessons learned from implementation and community response. This gives the reader context about the environment the feature exists in, deepening their understanding.

The final section synthesizes the information from the descriptions. When different communities have similar concerns it will merge these concerns into a single description. When the concerns are distinct it will clarify the distinction. It will also look for opportunities to apply solutions created by one community to a concern raised by another. Finally, it will provide a compact table listing all concerns. The mechanism will either address each of these concerns or provide justifications for leaving specific concerns unaddressed.

[6] It should go without saying that the author is also a reader, albeit a particularly invested one. =)

3 Python

Python provides the following features:

- Positional Value
  - Callers typically decide between naming or positioning values but authors can restrict this decision.
  - Positional values must precede named values.
- Default Value
- Named Value
- Typed Value
- Caller Destructuring
  - Restricted to iterables and dictionaries.
- Variadic Functions
  - One or two variadic variables will be bound to a list for positional values and/or a dictionary for named values.

Python used to provide author destructuring for tuples, but this was removed in version 3.0.

3.1 Background

Python has a compiler which produces bytecode [4, internals/compiler.rs section "Abstract"], and an interpreter which executes the bytecode [4, internals/interpreter.rst section "Introduction"]. Argument and return values are given using a stack managed by the interpreter; instructions may modify or move these values, even if this is not the primary purpose of the instruction [4, internals/interpreter.rst section "The Evaluation Stack"].

There are 2 families of instructions that are used throughout the examples in this section. The `LOAD` family puts values on the stack from different locations depending on the instruction. The `CALL` family initiates a function call after the stack has been prepared. There are also example-specific instructions

which will be explained alongside the relevant example.

Note: This section omits bookkeeping instructions that are not topically relevant. For example, when calling a non-method function (one which is not associated with an object instance), the interpreter pushes `NULL` *onto the stack before pushing the function. This instruction, and similarly uninteresting instructions, are omitted for brevity.*

3.1.1 `LOAD` family

Instructions prefixed with `LOAD_` retrieve a value from some location (depending on the instruction) and put it onto the stack. Each instruction recieves an integer which represents an index into a C-level array. Which array is referenced depends on the instruction. When the array contains variable names, the instruction also retrieves the value associated with that name. The below table explains the contents of the array that each instruction references.

`LOAD_CONST` Constant values which appear literally or implicitly in source code [3, Doc/library/dis.rst lines 964-966].

`LOAD_FAST` Names of local variables which are guaranteed to be initialized. [3, Doc/library/dis.rst lines 1253-1259, Lib/inspect.py line 514]

`LOAD_NAME` Names of non-local variables. If a local variable exists with the same name as a non-local variable then the value bound to the local variable will be returned. [3, Doc/library/dis.rst lines 969-972, Lib/inspect.py line 511].

3.1.2 `CALL` Family

These instructions tell the interpreter to call a function. This paper views this family as rooted in the plain `CALL` instruction, with all others being variants on this core instruction. When it needs to reference a behavior which occurs when a function is called, it examines only the plain `CALL` instruction and assumes that other instructions behave similarly unless the purpose of the variant is to change that specific behavior.

`CALL` Receives an integer indicating the number of argument values provided by the caller. The stack will contain the function to call followed by the argument values in separate stack locations. [3, Doc/library/dis.rst lines 1398-1410, Python/ceval.c lines 1314-1536].

3.2 Simple Function and Call

```
def add_values(a, b):
    return a + b

add_values(1000, 1001)
```

The bytecode generated for the call to `add_values` performs 3 tasks. First, it pushes the function onto the stack. Next, it pushes literal values which will become associated with arguments. Finally, it calls the function.

```
LOAD_NAME  0 (add_values)
LOAD_CONST 1 (1000)
LOAD_CONST 2 (1001)
CALL       2
```

The bytecode generated for the definition is similar. It uses `LOAD_FAST` (instead of `LOAD_NAME`) to refer to the variables associated with arguments and `BINARY_OP` (instead of `CALL`) to use the built-in `+` operator.

```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
BINARY_OP    0 (+)
```

3.3 Positional Value

Provided by default. Generally, callers can choose whether to provide values by name or position when they make the call. All positional values must precede all named values [6, reference/expressions.html section 6.3.4 "Calls"]. Function authors can specify that some arguments with only receive their value by position. These are referred to as "positional-only arguments".

3.3.1 Syntax & Semantics

Callers specify positional values by providing a comma-separated list of values. Function authors specify positional-only arguments by listing a literal `/` after the final positional-only argument [6, reference/compound_stmts.html section 8.7 "Function Definitions"]:

```
def add_values_mixed(position, /, either):
    return position + either

# valid
add_values_mixed(1000, 1001)
add_values_mixed(1000, b=1001)

# invalid: the value for argument
# "position" cannot be given by name
# add_values_mixed(position=1000, \
#                  either=1001)
```

3.3.2 Implementation

The bytecode for function definitions is identical to the simple definition regardless of the presence of positional-only arguments. The restriction is enforced within the `CALL` instruction. In particular, the helper function `positional_only_passed_as_keyword` uses the `co_posonlyargcount` and `co_localsplusnames` members of the code object. These variables track the number of positional-only arguments [3, Doc/library/inspect.rst lines 180-181] and the names of all arguments [3, Include/cpython/code.h line 155] respectively. If any names of positional-only arguments appear as keyword arguments then the helper raises an error. [3, Python/ceval.c lines 1182-1244]. Note that the helper is only called if the function does not include a variadic variable for named values [3, Python/ceval.c lines 1417-1431].

The bytecode for positional values is identical to the bytecode for the simple call. Python stores the values of local variables in the C-level array 'localsplus'. The `CALL` instruction copies positional arguments from the stack into this array [3, Python/ceval.c lines 1341-1353].

3.3.3 Historical Record

Positional values have always been available in Python and requires no special syntax to use.[7]

PEP 570 introduced positional-only arguments [5, peps/pep-0570.rst]. It gives several justifications for the change, most of which are concerned with maintaining a healthy ecosystem. There are two relevant [8] ecosystem harms the PEP is concerned with: inappropriate use of value names by callers and increased maintenance burden for library authors.

Inappropriate use of value names includes using non-meaningful names, such as a math function that takes one argument (the `sqrt` function takes one argument named `x`). It also includes providing values in an illogical order, such as calling the `range` function and supplying the `stop` value before the `start` value.

The increased maintenance burden occurs because all argument names are automatically and irrevocably added to the API surface of all libraries. It could be the case that a library author wants to implement a change which should be non-breaking in principle, but prompts a variable name change for clarity. This variable name change transforms the overall change into a breaking change.

The PEP is also concerned with functions that include a variadic variable for named parameters. For these functions, any non-variadic variables restrict the domain of the variadic variable, as their names will be associated with the distinct variable rather than the variadic one.

Finally, the PEP notes the curious case of the `range` function, which the PEP describes as accepting "an optional parameter to the left of its required parameter." In particular, if the `range` function only receives a single argument it is interpreted as the end

[7] Unfortunately, positional values are assumed to be the default method of passing arguments by most programmers, including language authors, so there is no good citation for this assertion.

[8] There are also several concerns mentioned which are specific to Python and/or its implementation. For example, it references PEP 399 which requires that pure Python code and C extensions have the same expressive power. While important to the Python community, these concerns are not relevant to this paper.

of the range, but if it receives 2 arguments then the first is interpreted as the start while the second is interpreted as the end. This concern does not appear to be addressed by PEP 570 [9].

3.4 Default Value

Provided by default argument values.

3.4.1 Syntax & Semantics

Function authors can define a default value by adding a literal `=` after the name of an argument, then the value [6, reference/compound_stmts.html section 8.7 "Function Definitions"].

```
def add_values(mandatory, optional=2000):
    return mandatory + optional

# valid
add_values(1000)
add_values(1000, 1001)

# invalid: the first argument is not
# associated with a default value
# add_values()
```

3.4.2 Implementation

Default values do not impact the bytecode generated for the definition or the call: they are both identical to the simple call. Instead, the `CALL` instruction retrieves default values from the code object and uses them when necessary [3, Python/ceval.c lines 1314-1536, trace_call_TO_initialize_locals].

3.4.3 Historical Record

Default values were added in version 1.0.2 [3, Misc/HISTORY lines 32809-32811]. There is additionally a note that default values "would now be quite sensible" in the version 0.9.4 release notes. This version changed argument processing so that functions receive all arguments as separate values, rather than as a single tuple [3, Misc/HISTORY lines 34550-34639].

PEP 671 proposes adding a feature which would allow function authors to provide an expression which will produce a default value at call time ("late evaluation") [5, peps/pep-0671.rst]. Currently, default values must be static/constant values which are determined when the function is defined. The mailing list discussion includes several disagreements, including whether or not it is appropriate for a function signature to contain un-inspectable objects and technical difficulties about scoping rules for late evaluated values. The proposal is still in the "draft" state, so it might be added in the future (possibly after trivial or significant changes to the proposal), but there has been no activity on the mailing list since 2021. [9]

3.5 Named Value

Provided through keyword arguments. Generally, callers can choose whether to provide values by name or position when they make the call. All named values must proceed all positional values [6, reference/expressions.html section 6.3.4 "Calls"]. Function authors can specify that some arguments will only receive their value by name. These are referred to as "keyword-only arguments".

3.5.1 Syntax & Semantics

Callers provide named values by writing first a symbolic name, then a literal `=`, then the value. [6, reference/expressions.html section 6.3.4 "Calls"]. Function authors specify keyword-only arguments by listing a literal `*` before the first keyword-only argument. [6, reference/compound_stmts.html section 8.7 "Function Definitions"].

```
def add_values_mixed(either, *, named):
    return either + named

# valid
add_values_mixed(named=1001, either=1000)
```

[9] At least, I do not see anything that addresses it when I read the PEP and the implementation of range still inspects the number of provided arguments manually [3, Objects/rangeobject.c lines 81-120].

```
add_values_mixed(1000, named=1001)

# invalid: the value for argument "named"
# must be given by name
# some_function(1000, 1001)

# invalid: named values cannot appear
# before positional values
# some_function(named=1001, 1000)
```

3.5.2 Implementation

The bytecode for function definitions is identical regardless of the presence of keyword-only arguments. The restriction is enforced within the `CALL` instruction. In particular, the helper function `initialize_locals` checks that the number of positional arguments is not more than expected [6, Python/ceval.c lines 1458-1462, trace_call_TO_initialize_locals], by checking the `co_argcount` member which tracks the number of arguments which may be positional [3, Doc/library/inspect.rst lines 146-149]. [3, Python/assemble.c line 556].

Bytecode for calls which use named values differ significantly from the simple call. For example, consider this code:

```
def add_values(a, b):
    return a + b

add_values(b=1001, a=1000)
add_values(1000, b=1001)
```

The bytecode for the first call differs from the simple call by adding the `KW_NAMES` instruction prior to the `CALL` instruction:

```
LOAD_NAME                1 (add_values)
LOAD_CONST               6 (1001)
LOAD_CONST               5 (1000)
KW_NAMES                 8 (('b', 'a'))
CALL                     2
```

`KW_NAMES` marks the given constant, in this case the tuple `('b', 'a')`, as a set of argument names to be used by `CALL` [3, Python/bytecodes.c lines 2601-2605, 2644, 2869–2692, 2706–2709]. Then, the `CALL` instruction determines which values belong to which arguments by corresponding their respective positions on the stack and in the tuple. [3, Python/ceval.c lines 1383-1384].

The process is similar when some values are provided by position and others by name. The second call above does not have any additional instructions to handle this case:

```
LOAD_NAME                2 (add_values_mixed)
LOAD_CONST               5 (1000)
LOAD_CONST               6 (1001)
KW_NAMES                12 (('named',))
CALL                     2
```

The `CALL` instruction infers which value is named based on the restriction that positional values must precede named values [3, Python/ceval.c lines 1383-1384].

3.5.3 Historical Record

Named values were first introduced in Python 1.3 [2, Doc/tut.tex lines 3540-3626]. The feature was based on the similar feature provided by Modula-3 [2, Doc/tut.tex lines 3584-3586]. While keyword-only arguments (discussed later in this section) were added afterwards, the core syntax and semantics of keyword-only arguments have remained unchanged.

PEP 3102 introduced keyword-only arguments. It provides a single justification for the change: variadic functions cannot make use of default values. The PEP gives the following example:

```
def sortwords(*words, case_sense=False):
    pass
```

If the value associated with `case_sense` can be provided positionally then it must be provided in every call even if the caller wants the default value of `False` [10].

[10] Unless the caller wants to sort the empty list. =)

3.6 Implicit Value

This feature is not provided by Python.

3.7 Typed Value

Provided through type hinting. The Python compiler and interpreter do not change their behavior based on type hints. However, they do guarantee that the hints will be available to external tools and provide supporting infrastructure to help the tools work correctly. Both static analyzers and runtime checkers can make use of annotations.

3.7.1 Syntax & Semantics

Type hints are specified using function annotations, as defined in PEP 3107. This means that function authors add a colon and type name after the variable name associated with the argument:

```
def typed_argument(x: str):
  pass
```

3.7.2 Implementation

Annotations are stored as metadata in the Python object which represents the function. Libraries can access them through the `__annotations__` property, which contains a dictionary [5, peps/pep-3107.rst, "Accessing Function Annotations"].

3.7.3 Historical Record

The foundations for type hinting were added in PEP 3102, which defines the syntax for function annotations [5, peps/pep-3102.rst]. The Python developers then waited for external community-driven tools to experiment with different type-checking approaches. Eventually, they took lessons learned from the community and created a set of recommendations in PEPs 482, 483, and 484 [5, peps/pep-0484.rst, "Abstract"]. Much of their content addresses type theory issues, such as generics, variance, and special types like `Any`. Since this initial introduction there have been a number of PEPs which further clarify best practices or provide syntactic improvements to type specifications. However, the core mechanism that this paper is concerned with - associating a type with an argument, regardless of how that type is specified - remains unchanged.

3.8 Caller Destructing

Provided through argument unpacking. Caller destructuring is only available for iterables and mappings.

3.8.1 Syntax & Semantics

This feature allows a caller to prefix one or more iterables with `*` in order to translate their contents into a set of positional values, and/or prefix one or more mappings with `**` to translate their contents into a set of named values. For mappings, keys must strings naming an argument. [6, reference/expressions.html section 6.3.4 "Calls"]

```
def add_values(a, b, c):
    return a + b + c

# all of the below calls are equivalent
# to this:
# add_values(1000, 1001, 1002)

# destructure an iterable into positional
# values
l = [ 1000, 1001, 1002 ]
add_values(*l)

# destructure multiple iterables into
# positional values
first_part = [ 1000 ]
second_part = [ 1001, 1002 ]
add_values(*first_part, *second_part)

# destructure a mapping into named
# values
d = { 'a': 1000, 'b': 1001, 'c': 1002 }
add_values(**d)

# destructure multiple mappings into
# named values
```

```
first_part = { 'b': 1001 }
second_part = { 'a': 1000, 'c': 1002 }
add_values(**first_part, **second_part)
```

3.8.2 Implementation

The difference between the simple call and a call which includes destructuring is best explained by starting with the final instruction. While the simple call uses the plain `CALL` instruction a destructuring call uses the `CALL_FUNCTION_EX` instruction. `CALL_FUNCTION_EX` receives either 0 or 1 which tells it whether or not there is a mapping to destructure [3, Doc/library/dis.rst lines 1398-1410].

When it receives 1, there is a mapping to destructure which will be on the top of the stack. While the caller can use any mapping (and any number of mappings), `CALL_FUNCTION_EX` will always see a single dictionary when it executes (the process which ensures this is discussed in more detail later in this section). The dictionary is turned into a set of keyword arguments by interpreting the keys as names identifying arguments. [3, Objects/call.c lines 1029-1053]

The next item on the stack is an iterable to destructure. In this case, `CALL_FUNCTION_EX` might see any iterable on the stack. If the iterable is not a tuple it will convert it into a tuple [3, Python/bytecodes.c lines 3198-3207]. The elements of this tuple will be used as positional values. [3, Python/bytecodes.c line 3219].

When `CALL_FUNCTION_EX` receives 0 the process is similar, except that the top element of the stack is an iterable and there is no mapping.

The compiler ensures that `CALL_FUNCTION_EX` only receives dictionaries (rather than the arbitrary mapping object provided by the caller) with two instructions. First, it issues a `BUILD_MAP` instruction to place a new dictionary on the stack [3, Doc/library/dis.rst lines 1015-1023]. Then it adds the keys and values of each mapping object into this dictionary by repeatedly calling the `DICT_MERGE` instruction. For example, this code:

```
add_values(*d)
```

Compiles to this bytecode (note that `BUILD_MAP` receives the value 0 to indicate that it is building an empty dictionary):

```
LOAD_NAME          0 (add_values)
LOAD_CONST        13 (())
BUILD_MAP          0
LOAD_NAME          3 (d)
DICT_MERGE         1
CALL_FUNCTON_EX    1
```

When named values are provided separately from the destructured values, the freshly created dictionary is prepopulated with those values. For example, this code:

```
d = { 'b': 1001, 'c': 1002 }
add_values(a=1000, **d)
```

Compiles to this bytecode (note that in this case, `BUILD_MAP` receives the value `1` to indicate that there is one key-value pair on the stack):

```
LOAD_NAME          1 (add_values)
LOAD_CONST        17 (())
LOAD_CONST        11 ('a')
LOAD_CONST         3 (1000)
BUILD_MAP          1
LOAD_NAME          4 (d)
DICT_MERGE         1
CALL_FUNCTION_EX   1
```

When the caller provides multiple destructured iterables, or provides literal positional values in addition to one or more destructured iterables, the compiler issues instructions to merge them into a list, then converts that list into a tuple. For example, this code:

```
t0 = ( 1001, )
t1 = ( 1002, )
add_values(1000, *t0, *t1)
```

Compiles to this bytecode:

```
LOAD_NAME          1 (add_values)
LOAD_CONST         3 (1000)
```

```
BUILD_LIST       1
LOAD_NAME        4 (t0)
LIST_EXTEND      1
LOAD_NAME        5 (t1)
LIST_EXTEND      1
CALL_INTRINSIC_1 6 (INTRINSIC_LIST_TO_TUPLE)
CALL_FUNCTION_EX 0
```

If the caller provides only a single iterable to destructure, and no literal positional values, this iterable is placed onto the stack without modification and the tuple creation logic contained within `CALL_FUNCTION_EX` itself is triggered.

3.8.3 Historical Record

When argument unpacking was first introduced in version 1.6 [3, Misc/HISTORY lines 26740-26743], it only allowed callers to unpack a single iterable and/or a single mapping. For example, the call `add_values(*first_part, *second_part)` would have been illegal. PEP 448 expanded argument unpacking so that multiple values can be destructured in the same call [5, peps/pep-0448.rst]. The rationale given for this change was enhanced readability, as previously callers would either need to build iterables/dictionaries separately or destructure them manually, adding additional lines of code which are semantically sparse.

3.9 Author Destructuring

While python does not currently support authorial destructuring, it did so prior to version 3.0 [5, peps/pep-3113.rst]. It allowed authors to declare that arguments should receive tuples whose values would be bound to separate local variables:

```
def distance((x1, y1), (x2, y2)):
  pass
```

This function would require that callers pass in 2 values which are both tuples containing 2 elements. The values from the first tuple would be bound to local variables `x1` and `y1`, while the values from the second would be bound to `x2` and `y2`.

The functionality was removed through PEP 3113. The rationale includes multiple implementation issues which are important to the Python community but not relevant to this paper.

3.10 Variadic Function

Provided by arbitrary argument lists and dictionaries. Positional values are collected by the former while named values are collected by the latter.

3.10.1 Syntax & Semantics

Function authors specify variadic-ness by specifying the name for one or two variadic variables. The name for the variadic list must be prefixed by a `*` while the name for the variadic dictionary must be preceeded by a `**` [6, reference/compound_stmts.html section 8.7 "Function Definitions"].

```
from itertools import chain

def add_values(*pos_vals, **named_vals):
    return sum(chain(pos_vals, \
                     named_vals.values()))

# All of these values appear in the
# pos_values list
add_values(1000, 1001, 1002, 1003)

# All of these values appear in the
# named_values dictionary
add_values(named_val0=1000,
           named_val1=1001,
           named_val2=1002,
           named_val3=1003)

# The values 1002 and 1003 appear in the
# pos_vals list while the names and
# values named_arg0=1000 and
# named_arg1=1001 appear in the
# named_vals dictionary
add_all_values(1002,
               1003,
               named_val0=1000,
```

```
named_val1=1001)
```

3.10.2 Implementation

The interpreter tracks which positional values are also variadic values by checking the `co_argcount` variable associated with the function's code object. Remaining positional arguments are moved into the appropriate variadic variable, if it exists [3, Python/ceval.c lines 1355-1376]. It distinguishes variadic named values from non-variadic named values by checking if the name is expected; the interpreter already has to keep track of this information because an unrecognized value name is considered an error for non-variadic functions [3, Python/ceval.c lines 1378-1455].

3.10.3 Historical Record

PEP 468 updated the variadic variable for named values such that the author can retrieve the syntactic order in which the values were given. The rationale given for this change is that some users are developing APIs where order matters, such as serialization. [5, peps/pep-0468.rst]

4 Guile

Guile provides the following features:

- Destructuring
- Named Value
- Default Values
- Positional Value
- Typed Values
- Variadic Functions

All code samples were compiled with the partial-evaluation optimization turned off. This is because partial evaluation frequently removes the function call itself as the return values can be calculated at compile-time.

Additionally, bytecode listings include comments which help explain what the instruction is doing. The meaning of the comment depends on the instruction. For example, the `make-immediate` bytecode instruction copies a literal static value onto the stack. It contains a comment which indicates the value loaded:

```
(make-immediate 4 4002)          ;; 1000
```

While the `call-scm<-scm-scm`, which is pronounced "Call scheme from scheme scheme" to signify that it is calling a built-in function that returns a scheme value and accepts two scheme values as input, includes the name of the built-in function it is calling:

```
(call-scm<-scm-scm 8 8 7 111)    ;; lookup-bound
```

Note that the `alloc-frame` instruction will contain a comment referring to the number of "slots" that the frame has; this refers to the size of the stack after the instruction finishes executing.

These comments are helpfully added by the Guile decompiler.

4.1 Background

Guile implements the Scheme programming language, which is a dialect of Lisp. Scheme was originally described in a 1975 paper for demonstrative purposes [17, page 1]. Interest in the language led to a series of revisions to the original description and, eventually, standardization. These papers are refered to as "Revised Reports on Scheme", abbreviated to "$r^N rs$" where the N is replaced with a number representing the revision count. For example, the most recently published version of the standard is the 7th revision of the Scheme programming language so it is referred to as "$r^7 rs$".

Guile was originally implemented as an interpreter which worked with a literal representation of a program's text [15, Section 9.3.1 "Why a VM?"]. A virtual machine was added to Guile in the 2.0 release (2010) [15, Section 9.1.4 "A Timeline of Selected Guile Releases"] [12, NEWS lines 5068-5071], and was rewritten for the 2.2 release (2017) [15, Section 9.1.4 "A Timeline of Selected Guile Releases"]. Modern Guile implements a compiler which produces bytecode, an interpreter which executes bytecode, and an interpreter which directly executes program text [12, modules/language/scheme/spec.scm lines 43-45]. This paper will focus on compiled bytecode as this keeps the analysis consistent with other sections and is the typical way to execute Guile code[11].

Guile bytecode operates as a stack machine with 2 pointers into the stack: the frame pointer and the stack pointer. [15, Section 9.3.2]. The frame pointer stores a location near[12] the beginning of the frame, where each frame represents a single function call. The stack pointer keeps track of the end of the stack, like a pointer to the end of an array [15, Section 9.3.3]. When instructions take an index, they differ in whether they take in an index relative to the stack pointer or the frame pointer [15, Section 9.3.5]. This paper will always reference indexes relative to the stack pointer for the sake of consistency; this has the effect that frames start at a higher index and end at a lower index. For example, in the simple call the frame pointer, associated with the beginning of the stack, is moved to index 5 while the stack pointer, associated with the end of the stack, is moved to index 2.

Guile produces different kinds of call instructions based on the call's location. Code samples in the repository were intentionally crafted to ensure that they always produced the plain `call` instruction; irrelevant parts of the code (such as a constant value placed after the call) are omitted from this paper.

4.1.1 The optargs module

Many of the features described in this paper are provided through Guile's `optargs` module, which was introduced in version 1.3.2 (1999) [15, Section 9.1.4 "A Timeline of Selected Guile Releases"] [12, NEWS lines 11451-11525]. At this time, Guile lacked a virtual machine and the module was implemented in pure scheme. This implementation operated by adding a prelude to the function definiton which searches through the values provided by the caller to decide which values should be assigned to which variables [10, ice-9/optargs.scm]. The virtual machine was added to Guile in version 2.0.0 (2010) [15, Section 9.1.4 "A Timeline of Selected Guile Releases"] [12, NEWS lines 5068-5071]. This also entailed a rewrite of the `optargs` module which centered the implementation around the internal `<lambda-case>` strucure. This structure contains all the information about the different kinds of arguments accepted by the function and facilitates the use of special-purpose VM instructions for processing arguments efficiently [14, 2009-10 lines 6428-6453]. While the VM has gone through changes since then, including a complete rewrite in version 2.2 [15, Section 9.1.4 "A Timeline of Selected Guile Releases"], the implementation of the `optargs` module has remained stable. The special-purpose VM instructions have been adapted to handle the details of VM operation correctly and decrease latency, but the core logic used to process arguments has proven to be robust.

[11] For example, running `guile script-name` will first compile the script then run the compiled file rather than running the script directly.

[12] It is "near" the beginning rather than "at" the beginning because some metadata is stored before the frame pointer's location; this metadata will not be relevant to this paper.

4.2 Simple Function and Call

```
(define (add-values a b)
  (+ a b))

(add-values 1000 1001)
```

The bytecode generated for the function definition is fairly straightforward.

```
0 (call-scm<-scm-scm 1 1 0 0)      ;; add
1 (reset-frame 1)                  ;; 1 slot
2 (handle-interrupts)
3 (return-values)
```

Instruction 0 calls the built-in function `add` (which corresponds to the `+` function) and places the result into index 0 [12, liguile/vm-engine.c lines 1545-1549].

Instruction 1 resizes the stack so that it contains 1 element - in this case, the return value [12, libguile/vm-engine.c lines 797-802].

Instruction 2 ensures that Guile properly handles signals like Ctrl-C (SIGINT) and code instrumentation [15, Section 9.3.7.6 "Instrumentation Instructions", Section 6.22.3 "Asynchronous Interrupts"]; it can be safely ignored here and it will not be mentioned again.

Finally, the `return-values` instruction moves the flow of execution back to the caller. Counterintutively, it does not actually handle moving the return valuse to specific locations; it only sets the frame and instruction pointers to the caller's values [12, libguile/vm-engine.c lines 530-555].

The bytecode generated for the call to `add-values` is more complicated because it is calling a user-defined function. It needs to load the function and its arguments [13], then dispatch to the function.

```
0 (static-ref 7 16324)             ;; add-values
1 (call-scm<-scm-scm 8 8 7 111)    ;; lookup-bound
2 (scm-ref/immediate 5 8 1)
3 (make-immediate 4 4002)          ;; 1000
4 (make-immediate 3 4006)          ;; 1001
5 (handle-interrupts)
6 (call 3 3)
```

Instruction 0 loads the literal string `"add-values"` into index 7 [12, libguile/vm-engine.c line 2125-2129].

Instruction 1 looks up the value associated with the name `add-values` and stores it in index 8 [12, libguile/vm-engine.c lines 1545-1549, libguile/intrinsics.c lines 374-376].

The value includes both the procedure itself and some associated metadata; instruction 2 retrieves the procedure itself and stores it in index 5 [12, libguile/vm-engine.c lines 1906-1910].

Instructions 3-4 place the constant values 1000 and 1001 into indices 4 and 3, respectively. At this point the most relevant portion of the stack looks like this:

Index	Value
5	#<procedure add-values>
4	1000
3	1001

Finally, instruction 6 calls the function; internally, this involves moving the frame pointer to index 5 (where the procedure is stored) and the stack pointer to index 2 (the new end of the stack, which is smaller for the callee than for the caller) [12, libguile/vm-engine.c lines 451-454].

4.3 Caller Destructuring

Provided through the `apply` function. This allows a caller to provide a list which is destructured into a set of positional arguments.

4.3.1 Syntax & Semantics

Destructuring in guile has a simple interface, as `apply` is simply a function that takes in a function as the first value, then any number of arbitrary values, and requires a possibly empty list as the final value. For example, all of these are equivalent:

[13] The function definition did not have to manage the arguments for the built-in call because built-in calls take values from arbitrary stack locations which are specified in the instruction arguments, so the caller-supplied locations were reusable.

```
(add-values 3 4)
(apply add-values 3 '(4))
(apply add-values '(3 4))
```

However, this would be an error because the final value is not a list:

```
(apply add-values 3 4)
```

4.3.2 Implementation

`apply` operates by first collecting all of the positional arguments into a single list, with the final value as the tail of the list, and sending them to `scm_call_n` [12, libguile/eval.c lines 585-622, lines 715-729] [14]. This function is responsible for setting up the VM stack correctly [12, libguile/vm.c lines 1542-1621]. For example, these lines save the frame's metadata and add the arguments to the correct stack positions:

```
SCM_FRAME_SET_VIRTUAL_RETURN_ADDRESS (call_fp, vp->ip);
SCM_FRAME_SET_MACHINE_RETURN_ADDRESS (call_fp, 0);
SCM_FRAME_SET_DYNAMIC_LINK (call_fp, return_fp);
SCM_FRAME_LOCAL (call_fp, 0) = proc;
for (i = 0; i < nargs; i++)
  SCM_FRAME_LOCAL (call_fp, i + 1) = argv[i];
```

These tasks are handled by different components in the simple call. The frame's metadata is typically set by the `call` instruction , while the arguments are added to the stack by the by separate instructions such as `make-immediate` or `scm-ref`.

[14] A typical guile programmer (such as myself) who is familiar with the dotted list syntax or the `#:rest` argument (both described in the section on variadic functions) might be confused by the C-level API of the apply function. It takes 2 required arguments and a rest argument. The apply function takes any number of arguments and requires that the final argument is a list. Oddly, the body of the function simply calls `cons*` on the second required argument and the rest argument; one might expect that this would cause the user-provided list to be preserved instead of flattened. However, when a C-level function is registered with a rest argument in guile it receives an improper list of arguments, rather than the proper list provided when `#:rest` is given to `define*`. See the file `examples/guile/c-gsubr-rest.scm` (and its corresponding c file) in the paper's repository for a demonstration. The reason for this discrepency is not clear but the API of `scm_apply` makes sense now.

4.3.3 Historical Record

The notion of list destructuring with `apply` was included in the original paper defining the lisp programming language [8, pages 189-190]. This original definition took exactly 2 parameters, a function and a list containing the arguments. r^{2}rs extended this by allowing the caller to pass any number of values between the function and the list [16, page 57].

In the original interpreter, Guile simply called the function after manually unpacking the arguments [13, libguile/eval.c lines 1869-1985]. With the release of Guile 2.0, which added the virtual machine [12, NEWS lines 5068-5071], the `apply` function delegated to `scm_call_with_vm`, a dedicated C-level helper function for calling scheme-level functions [11, libguile/eval.c lines 782-802]. This was replaced with the `scm_call_n` function as part of a refactoring which removed explicit references to the VM from many locations in C code as well as eliminating VM visibility from scheme code [12, commit 165d9bf3a3bf34b53ed916743c6414f8030320c3 and earlier commits found with `git log libguile/eval.c`]. The reason for this change is not explicitly stated in the commit messages but is likely related to performance improvements attributed to a rewrite of the virtual machine in the Guile 2.2 release notes [12, NEWS lines 1995-2004, 2124–2205].

4.4 Author Destructuring

This feature is not provided by Guile.

4.5 Implicit Value

This feature is not provided by Guile.

4.6 Named Value

Provided through the `#:key` arguments to `optargs` definitions.

4.6.1 Syntax & Semantics

The caller must use the name specified by the author, which will also be the name used by the local variable

[15]. When an author mandates a named value they also implicitly create a default value. Details about default values are described in the appropriate section; for the purpose of named values all we need to know is that if a caller omits a value then the local variable will be bound to `#:false`. For example, an author of `add-values` could allow named values like this:

```
(define* (add-values #:key a b)
  (+ (or a 0) (or b 0)))
```

This function could be called in any of the following ways, with the result shown in the proceeding comment:

```
(add-values)              ; 0
(add-values #:a 3)        ; 3
(add-values #:b 3)        ; 3
(add-values #:a 3 #:b 3)  ; 6
```

However, these would not be legal:

```
(add-values 3)
(add-values 2 #:b 2)
```

When an author specifies that a value *can* be named this also means that it *must* be named, if it is provided at all.

Additionally, an author may specify that a caller can supply arbitrary named values which are not specified in the function signature (and not bound to a local variable, unless the function is also variadic). This is done by adding `#:allow-other-keys` [16] to the function signature. For example, a definition of `add-values` with this feature would allow all of the following calls to be legal, and they would all produce the value 6.

```
(define* (add-values #:key a b #:allow-other-keys)
  (+ (or a 0) (or b 0)))

(add-values #:a 3 #:b 3)
(add-values #:a 3 #:b 3 #:c 34)
(add-values #:c 34 #:a 3 #:b 3)
(add-values #:a 3 #:b 3 #:multiplier 88)
```

4.6.2 Implementation

When Guile compiles a procedure it stores the list of valid argument names next to the `SCM` object representing the code [12, libguile/vm-engine.c lines 734, 736, 743–745, libbugile/vm.c lines 1003, 1008–1009]. When a function using named values is defined, the compiler adds additional instructions as a prelude in the definition.

```
    0    (bind-kwargs 1 0 1 3 16351)
    1    (alloc-frame 3)                ;; 3 slots
    2    (immediate-tag=? 1 4095 2308)  ;; undefined?
    3    (jne 2)                        ;; -> L1
    4    (make-immediate 1 14)          ;; 3
L1:
    5    (immediate-tag=? 0 4095 2308)  ;; undefined?
    6    (jne 2)                        ;; -> L2
    7    (make-immediate 0 22)          ;; 5
L2:
    8    (call-scm<-scm-scm 2 1 0 0)    ;; add
    9    (reset-frame 1)                ;; 1 slot
   10    (handle-interrupts)
   11    (return-values)
```

Instruction 0, `bind-kwargs`, performs the bulk of the work required to process named values at runtime. First, it initializes all relevant local variables (eg, ones that have default values or are named) to the undefined value [12, libguile/vm.c lines 995-998]. Next, it walks through the list of non-positional arguments with the assumption that every even-indexed item is a keyword naming an argument and binds the odd-indexed items to the appropriate variables [17] [12, libguile/vm.c lines 1000-1037].

[15] The caller uses a keyword to name the value while the author uses a symbol to name the variable, as normal, but they both use the same name.

[16] Ironically, while the optargs module does not support implicit values for user-defined code, `#:allow-other-keys` is itself an implicit value.

[17] Technically, it assumes that alternating value are keywords, except that it will silently ignore a non-keyword if `#:rest` is included in the procedure definition. However, I am

Instruction 1 prepares the stack for the function call.

Instructions 2-7 ensure that the local variables which come from named values are initialized properly; these instructions are actually about default values, not named values, so they are discussed in the appropriate section.

Finally, instructions 8-11 are the body of the function as found in the simple definition.

The bytecode for the call is largely similar to the simple call, except that it also sends the keywords as additional argument values to the function (while they are not considered values within the body of a `define*` form, they are values at this lower level):

```
 1 (call-scm<-thread 8 62)          ;; current-module
 2 (static-ref 7 16324)             ;; add-values
 3 (call-scm<-scm-scm 8 8 7 111)    ;; lookup-bound
 4 (scm-ref/immediate 5 8 1)
 5 (static-ref 4 16331)             ;; #:a
 6 (make-immediate 3 34)            ;; 8
 7 (static-ref 2 16340)             ;; #:b
 8 (make-immediate 1 38)            ;; 9
 9 (handle-interrupts)
10 (call 3 5)
```

4.6.3 Historical Record

Named values were introduced with the original implementation of the `optargs` module [12, NEWS lines 11451-11525]. In the original implementation, the `let-ok-template` helper function generated a scheme-level `let` (or `let*`) form which initially binds each variable name to either the default value supplied by the author or a fresh undefined variable [12, commit 7e01997e88c54216678271de36b1c2088377492d ice-9/optargs.scm lines 128-135]. The `bindfilter` local in the `let-keyword-template` helper replaced these initial values with the values supplied by the caller [12, commit 7e01997e88c54216678271de36b1c2088377492d ice-9/optargs.scm lines 152-168].

currently focused on named values in isolation of other features. `#:rest` is discussed in the section on variadic functions.

The current implementation is largely the same, except that it takes advantage of being able to directly manipulate the VM state from C code instead of using macros to mutate code syntactically. The maintainers decided to move the implementation from scheme to C in order to decrease the latency associated with a function call that uses named values [14, 2009-10 lines 10053-10056, 14008–14009, 14139–14141].

One change, which may or may not be considered a bug fix, has to do with the way that Guile processes named values when the function also contains a variadic variable. Originally, Guile required that all named values precede all (other) variadic values. This was changed so that variadic values may be intermingled with named values, so long as variadic values are not keywords (unless `#:allow-other-keys` is set, in which case they may be keywords) [12, commit ff74e44ecba55f50b2c2c84bad2f13bed9489455].

4.7 Default Value

Provided through both `#:optional` and `#:key`. When considering default values, both of these are semantically equivalent. The difference between them is whether values are passed by position or by name. This section will exclusively use `#:optional` because none of the functions used in the examples are complicated enough to benefit from named values.

4.7.1 Syntax & Semantics

The default value for an argument can either be specified or unspecified. If it is unspecified it is `#false`. Arguments become associated with a default value by placing them after the `#:optional` keyword in a `define*` form. This verison of `add-values` demonstrates use of default values without specification.

```
(define* (add-values #:optional a b)
  (+ (or a 0) (or b 0)))
```

This function adds the given values, and the body of the function uses the value `0` if the caller omitted a vaule.

However, it is idiomatic to specify a default value in the function signature rather than testing the truthiness of the value. This is done by specifying the argument with a list instead of a symbol. The first element is the symbol naming the local variable and the second element is the default value.

```
(define* (add-values #:optional (a 0) (b 0))
  (+ a b))
```

This version is semantically equivalent to the first.

4.7.2 Implementation

The bytecode for the function definition differs significantly. I will start with the bytecode for the function which includes default values in the signature as this version is simpler:

```
   0    (bind-optionals 3)              ;; 2
   1    (alloc-frame 3)                 ;; 3
   2    (immediate-tag=? 1 4095 2308)   ;; u
   3    (jne 2)                         ;; -
   4    (make-immediate 1 2)            ;; 0
L1:
   5    (immediate-tag=? 0 4095 2308)   ;; undefined?
   6    (jne 2)                         ;; -> L2
   7    (make-immediate 0 2)            ;; 0
L2:
   8    (call-scm<-scm-scm 2 1 0 0)     ;; add
   9    (reset-frame 1)                 ;; 1 slot
  10    (handle-interrupts)
  11    (return-values)
```

Instruction 0, `bind-optionals`, checks if the caller omitted any argument values. If so, it fills in the associated variables with the undefined value [12, libguile/vm-engine.c lines 3213-3233].

The next 2 sections, instructions 2-4 and 5-7, check whether or not the local variables associated with optional values are undefined; if so, they are filled with the default value specified by the author.

Finally, instructions 8-11 are the body of the function as found in the simple call.

The version which does not specify default values in the signature is similar, but contains additional (and repetitious) logic.

```
 0 (bind-optionals 3)              ;; 2 argss
 1 (alloc-frame 3)                 ;; 3 slots
 2 (immediate-tag=? 1 4095 2308)   ;; undefined?
 3 (jne 2)                         ;; -> L1
 4 (make-immediate 1 4)            ;; #f
L1:
 5 (immediate-tag=? 0 4095 2308)   ;; undefined?
 6 (jne 2)                         ;; -> L2
 7 (make-immediate 0 4)            ;; #f
L2:
 8 (immediate-tag=? 1 3839 4)      ;; false?
 9 (jne 2)                         ;; -> L3
10 (make-immediate 1 2)            ;; 0
11 L3:
12 (immediate-tag=? 0 3839 4)      ;; false?
13 (jne 2)                         ;; -> L4
14 (make-immediate 0 2)            ;; 0
L4:
15 (call-scm<-scm-scm 2 1 0 0)     ;; add
16 (reset-frame 1)                 ;; 1 slot
17 (handle-interrupts)
18 (return-values)
```

Instruction 0 is also `bind-optionals`.

Instructions 2-7 fill in the default value but in this case the value is `#:false` rather than `0`.

The next two sections check the argument values a second time and replace the argument value with `0` if it was false (instructions 8-14) - these represent the `or` statements in the source code.

Instruction 15-18 are the body of the function as found in the simple call.

The bytecode for calling a function using default values does not differ from the bytecode for the simple call.

4.7.3 Historical Record

Default values were included in the original `optargs` implementation [12, NEWS lines 11451-11525]. The semantics at that time were nearly identical to the current semantics. The one notable difference is that, in the case where the author does not define a default value and the caller does not supply one, the variable was not bound to any value. This was changed as part of merging a branch that

was mostly concerned with how modules are handled. I was unable to find any record about the reason why this change was made [18] [12, commit 296ff5e78b8322fe4bf00c5ec1497dc28da776b8] [14, 2001-05 lines 11305-11313]. This was included in the 1.8 release series (2006) [15, Section 9.1.4 "A Timeline of Selected Guile Releases"] [19].

4.8 Positional Value

Provided as the default mechanism for mapping values to arguments.

4.8.1 Syntax & Semantics

The syntax and semantics for both calls and definitions are identical to the simple call.

4.8.2 Implementation

Guile does not explicitly enforce a restriction that values can only be supplied positionally. This is because, technically, all values are always supplied positionally. When a caller provides named values they are actually providing pairs of values: keywords as names and arbitrary values as values. For example, given this call it is impossible to say whether the caller is supplying named values or positional values:

```
(make-dictionary #:first  "one"
                 #:second "two"
                 #:third  "three")
```

This could be a valid call to a function with named values:

```
(define* (make-dictionary #:key first second third)
  (let ((result (make-hash-table)))
    (hash-set! result #:first  first)
    (hash-set! result #:second second)
    (hash-set! result #:third  third)
    result))
```

Or it could be a valid call to a function which takes 6 positional values:

```
(define* (make-dictionary key0 val0
                          key1 val1
                          key2 val2)
  (let ((result (make-hash-table)))
    (hash-set! result key0 val0)
    (hash-set! result key1 val1)
    (hash-set! result key2 val2)
    result))
```

From the perspective of this particular caller, it is impossible to determine which implementation is in use without inspecting the code [20].

However, it is likely that using named values for a function which does not accept them will result in an error. This is because names are themselves values and are unlikely to be the correct kind of value for the function. For example, this attempt to use named values resulted in an error because the local variable `a` is bound to the value `#:a`, which is not a valid input to `+`:

```
(define (add-values a b)
  (+ a b))

(add-values #:a 5 #:b 7)
; ice-9/boot-9.scm:1685:16: In procedure raise-exceptio
; In procedure +: Wrong type argument in position 1: #:
```

[18] Marius Vollmer, the developer who authored the commit, was very kind in responding to an email enquiring about this. Unfortunately it has been more than 2 decades since the change was made and it was not notable enough to warrant a place in permanent memory.

[19] I infer that it was first added in 1.8 because commit 296ff5e78b8322fe4bf00c5ec1497dc28da776b8, which implements the change, is between commit 0f24e75b73b9c0c745971de639a53748a395a1cb, which bumps the numbers in GUILE-VERSION from 1.7 to 1.9 (guile uses odd numbers for development versions and even numbers for release versions) and commit c299f186ff9693fc88859daef037e3d94cc7c0ff, which adds content to the NEWS file regarding changes new in 1.6 However, I did not find any content in the NEWS file which discusses the change (including in more recent release notes).

[20] Of course, a caller who wanted to use different keywords in their dictionary would notice the difference between these implmentations very quickly!

4.8.3 Historical Record

The Scheme standard requires that implementations accept value positionally. This has been true since the original paper describing lisp [8, pages 185-186]. The stated motivation is to distinguish between "functions" and "forms". In that paper, a function is the abstract idea of a formula that can have concrete numbers applied to it. Traditionally, a form is syntactically identical to a formula, but it is meant to be a stand-in for whatever value the formula will resolve to in context. The differentiating syntax is taken from a previous work which had a similar concern and does not explicitly justify the postional notation [1, pages 3-7] [21].

While the semantics of positional arguments have remained the same, the implementation has changed significantly. In the oldest version of Guile available through source control, values are given to a function by collecting them into a list, then dispatching based on the number of values expected by the function [13, libguile/eval.c lines 1822-2004]. With the addition of the VM, values are given by putting them into a frame managed by the VM (eg, not a C-level frame), avoiding the need to create a list unless the function is variadic.

4.9 Typed Value

This feature is provided by defining methods in the Guile Object-Oriented Programming System (GOOPS).

[21] The decision to supply arguments positionally appears to derive from prior work describing multi-argument functions as a series of functions which take in one of the values and call the "next" function which will receieve the "next" value (this is similar to the concept of currying used in some programming languages). With this model, it is natural to write values positionally as they must be provided in the correct order to ensure that each partial function operates correctly. It's also worth noting that the examples of functions used in the paper are relatively simple - they are all inlinable with the prose. In the face of such simplicity, naming values would be verbose without much benefit.

4.9.1 Syntax & Semantics

In GOOPS methods are not contained within classes. Instead, they are simply functions with type specifications. When an author defines a method, they can choose to specify a type by providing a two-element list in place of an argument name [15, Section 8.6 "Methods and Generic Functons"]. The first element of the list is the argument name and the second element is the class name. For example:

```
(define-method (add-values (a <number>) (b <number>))
  (+ a b))
```

This defines a method that will only be executed when it is called with exactly two arguments which are both numbers. The author can also define alternative implementations for different types[22]:

```
(define-method (add-values (a <list>) (b <list>))
  (append a b))
```

4.9.2 Implementation

[23]

Defining a method defines two separate functions: a generic which is responsible for method dispatching and the method which contains the body that the author defined [15, Section 8.6 "Methods and Generic Functons"]. A generic is an applicable struct

[22] While overloading is generally out of scope for this paper, the implementation section will not make sense without mentioning it.

[23] **TODO: Remove this footnote, this is by no means unique to this subsection and if it needs to be addressed it should be done so in an introductory section.** In most implementation sections I present the bytecode associated with a function definition and call and describe the low-level semantics. This would not be useful for this section. The GOOPS implementation is primarily concerned with function overloading, not with type checking. Most of the logic is related to checking the arity of the function and searching for the function with the "best fit" type signature (that is, it prefers more specific child classes over more general parent classes). The fact that it will produce an error if the user supplies invalid types appears to be incidental to the system, not a primary purpose. In order to avoid a drawn-out explanation which would not be relevant to this paper, I refer only to the source implementation which is relevant to type-checking.

(a structure which can be called like a function) which stores all methods which share a name [12, module/oop/goops.scm lines 2045-2245]. When a caller uses that name with some arguments, the generic object searches all of the methods associated with that name for a signature with the best fit [12, module/oop/goops.scm lines 1381-1449]. This indirectly causes an error if the caller provides a set of values which do not have a compatible type: the searching process will fail to find a match and print a message like this one:

```
ice-9/boot-9.scm:1685:16: In procedure raise-e
No applicable method for #<<generic> add-value
```

The bytecode for calling a function with typed values is identical to the simple call.

4.9.3 Historical Record

GOOPS was first added to the Guile repository in version 1.6 [15, Section 9.1.4 "A Timeline of Selected Guile Releases"]. Its implementation was based on STklos, the object system for STK, and it was also influenced by CLOS, the object system for Common Lisp [12, module/oop/goops.scm lines 24-25] [15, Section 8.0 "GOOPS"]. The original code [24] looks quite different from the modern version of the file due to changes in the core language, bug fixes, and refactorings for performance and readability. However, the underlying idea of storing all alternative implementations of a function in a structure that uses a type signature to implementation map has remained stable.

[24] I am tentatively using 4b5d86e0334f6b8c0b37c55cf47a4cd30e7a03e0 as the commit which "adds GOOPS". This is somewhat fictitious; 9 commits prior (fdd70ea97c142dc8db1e3f147ac6f5bd6ae157c6) changed the major version "due to the merge of GOOPS". However, at the time this commit was made no GOOPS files had been added; the version number was adjusted in anticipation of the merge, not after the merge was complete. I am choosing the other commit as representing the initial add of GOOPS to the repository because it is the last consecutive commit which adds or modifies a GOOPS file after the version adjustment.

4.10 Variadic Function

Provided by the "dotted list" syntax and the `#:rest` keyword in a `lambda*`.

4.10.1 Syntax & Semantics

The `#:rest` syntax allows an author to define a variadic variable that will contain a (possibly empty) list of variadic values. The variadic variable must be declared after all other declarations. For example, this definition and call would have the following value [25]:

```
(define* (add-values a b #:rest variadic-variable)
    (apply + a b variadic-variable)

(add-values 1000 2001 1002 2003)
; 6006
```

The dotted list syntax is similar, except that the author uses a single period instead of the `#:rest` keyword and this syntax works without `optargs`:

```
(define* (add-values a b . variadic-variable)
    (apply + a b variadic-variable)

(add-values 1000 2001 1002 2003)
; 6006
```

Other than this syntactic difference, the two methods of defining a variadic function are identical.

4.10.2 Implementation

For definitions, the only notable difference from the simple function is the addition of the `bind-rest` instruction near the beginning of the body. This instruction takes values from the stack and constructs a list which contains all of them; this list is bound to the variadic variable [12, libguile/vm-engine.c lines 756-783, libguile/vm.c lines 1041-1052].

[25] As the variadic variable contains a list, the below code uses `apply` to destructure the list. This is described in detail in the section on caller destructuring.

```
static SCM
cons_rest (scm_thread *thread, uint32_t base)
{
  SCM rest = SCM_EOL;
  uint32_t n = frame_locals_count (thread) - base;

  while (n--)
    rest = scm_inline_cons (thread, SCM_FRAME_LOCAL (thread->vm.fp, base + n),
                            rest);

  return rest;
}
```

For calls, the bytecode is identical to the simple call.

4.10.3 Historical Record

The dotted list syntax was first described in r^2rs, with the same semantics that are in use today [16, page 13]. The `#:rest` syntax was originally used by other lisp dialects and implementations of Scheme; Guile provided it in the original release of the optargs module in order to ease the transition for programmers used to those languages [12, commit 0a852b9424f949575afecb19a391023acc63e635 NEWS lines 849-843]

As described in the section on positional values, the oldest versions of Guile manually collected argument values into a Scheme-level list and dispatched based on the number of values given. This made it trivial for the interpreter to supply the variadic values by dropping elements from the front of the list of all arguments [13, libguile/eval.c lines 1652-1655].

When the VM was added in version 2.0, the `bind-rest` instruction was included. It performs essentially the same work as the current implementation [11, libguile/vm-i-system.c lines 718-732].

Part II
Analysis

Part III
Implementation

References

[1] Alonzo Church. "The Calculi of Lambda-Conversion". In: *Journal of Symbolic Logic* 6.4 (1941). DOI: 10.2307/2267126. URL: https://doi.org/10.2307/2267126.

[2] Python Software Foundation. *Python 1.3 Source Code*. https://legacy.python.org/download/releases/src/python-1.3.tar.gz.

[3] Python Software Foundation. *Python 3.12.2 Source Code*. https://github.com/python/cpython.

[4] Python Software Foundation. *Python Developer Guide*. https://github.com/python/devguide.

[5] Python Software Foundation. *Python Enhancement Proposals (PEPs)*. https://github.com/python/peps.

[6] Python Software Foundation. *Python HTML Documentation*. https://docs.python.org/3/archives/python-3.12.2-docs-html.zip.

[7] *Programming Languages – C*. Standard. International Organization for Standardization.

[8] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

[9] *PEP 671 (late-bound arg defaults), next round of discussion!* https://mail.python.org/archives/list/python-ideas@python.org/thread/UVOQEK7IRFSCBOH734T5GFJOEJXFCR6A/.

[10] GNU Project. *Guile 1.3.2 Source Code*. https://git.sv.gnu.org/git/guile.git.

[11] GNU Project. *Guile 2.0.0 Source Code*. https://git.sv.gnu.org/git/guile.git.

[12] GNU Project. *Guile 3.1.2 Source Code*. https://git.sv.gnu.org/git/guile.git.

[13] GNU Project. *Guile Source Code (oldest commit)*. `https://git.sv.gnu.org/git/guile.git`.

[14] GNU Project. *Index of /archive/mbox/guile-devel*. `https://lists.gnu.org/archive/mbox/guile-devel/`.

[15] GNU Project. *The Guile Reference Manual*.

[16] *Revised Revised Report on Scheme or An Un-Common Lisp*. Standard. Massachusetts Institute of Technology - Artificial Intelligence Laboratory.

[17] *Scheme: an Interpreter for Extended Lambda Calculus*. Standard. Massachusetts Institute of Technology - Artificial Intelligence Laboratory.