Snek: Overloading Python Semantics via Virtualization

JAMES M. DECKER[1,2], DAN MOLDOVAN[2], ANDREW A. JOHNSON[2], GUANNAN WEI[1], VRITANT BHARDWAJ[1], GREGORY ESSERTEL[1], FEI WANG[1], ALEXANDER B. WILTSCHKO[2], TIARK ROMPF[1] ([1]PURDUE UNIVERSITY, [2]GOOGLE BRAIN)

The Python language enjoys widespread adoption in a wide variety of domains spanning machine learning, scientific and high performance computing, and beyond. While implemented as libraries in Python, many Python frameworks aspire to be a domain-specific language (DSL), and often aim to bypass the Python interpreter. However, because Python's inherent ability to overload built-in structures such as declarations, conditionals, or loops is limited, these frameworks are often constrained to a suboptimal, API-centric interface that replicates these built-ins with framework-specific semantics. Such an approach ultimately yields productivity losses for programmers, especially when switching between or mixing frameworks, as users must memorize an ever-expanding list of method calls for performing even simple tasks such as constructing control flow. Furthermore, API designers are forced to create new, substituting abstractions for traditional programming constructs, forcing a steep learning curve for users.

In this paper, we propose a structured methodology to allow DSL developers to use the whole of Python as a front-end, rather than creating equivalent APIs or relying on shims. Our methodology provides an extensive operator overloading and virtualization mechanism through the use of source code transformations, and enables powerful mechanisms like type-based multi-stage programming (which is popular in statically typed languages), without requiring explicit type information (e.g., via type annotations).

We implement this methodology in a system called Snek, which represents the first type-driven multi-stage programming framework for a dynamic language which does not require extra-linguistic mechanisms, and demonstrate the ability to quickly and easily provide new semantics for Python constructs.

1 INTRODUCTION

Python has become one of the most widely-used programming languages, in part due to its high-level syntax and dynamic type system, and is currently the de facto language for numeric computing. Frameworks exist which enable programmers to use Python to target nearly every domain, with tools like Django [Django Contributors 2019] for web programming, PyTorch [PyTorch Contributors 2019] and TensorFlow [Abadi et al. 2016] for deep learning, Z3Py [Microsoft Research 2019] for SMT solving [de Moura and Bjørner 2008], Dask [Dask Contributors 2019] for distributed programming, and NumPy [van der Walt et al. 2011] for scientific computing.

However, despite this wide use, there exist a number of complications which cause difficulty for designers of these frameworks. Python's interpreter introduces nontrivial overhead [Al-Rfou et al. 2016], which runs counter to its role as a high performance numeric computing interface. While work has been done to bypass this interpretive overhead [Palkar et al. 2018], Python's limited operator overloading, counterintuitive scoping, and lexical rules [Politz et al. 2013] complicate such efforts. Indeed, such difficulties have led many of the above frameworks to rely almost entirely on API calls (e.g., `tf.cond` and `tf.while_loop` in TensorFlow or `z3py.If` in Z3Py), with users unable to express even simple control flow in idiomatic Python [Moldovan et al. 2018]. This is complicated further by Python's heavy reliance on mutation semantics. Python is also neither distributed nor

Author's address: James M. Decker[1,2], Dan Moldovan[2], Andrew A. Johnson[2], Guannan Wei[1], Vritant Bhardwaj[1], Gregory Essertel[1], Fei Wang[1],
Alexander B. Wiltschko[2], Tiark Rompf[1] ([1]Purdue University, [2]Google Brain).

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

inherently portable, preventing it from being run on embedded devices or on specialized hardware like GPUs or TPUs [Jouppi et al. 2017].

Current efforts to circumvent these difficulties in Python while maintaining syntax as close to idiomatic Python as possible vary in complexity, with some systems choosing to go so far as to modify or replace the Python interpreter entirely [Jeong et al. 2019; MacroPy3 Contributors 2019; RustPython Contributors 2019]. Others elect to bypass the Python interpreter altogether by adopting a wholesale translation strategy [Behnel et al. 2011; PyTorch Contributors 2018; van Merrienboer et al. 2018], and still others perform tracing (potentially as a translation strategy) [Frostig et al. 2018; PyTorch Contributors 2018, 2019]. Lastly, some systems [Moldovan et al. 2019] use source code transformation to virtualize a subset of Python in order to generate specialized code.

Each of these strategies provides significant benefits for the end user, though they come at a significant cost for the framework developer. Furthermore, the benefits of one strategy may not be captured in another, and moving between systems is not always feasible for end users. In this paper, we propose a structured methodology for source code transformations in Python in order to provide custom behavior for Python constructs at run time. We demonstrate the use of extensive operator overloading and virtualization of Python syntactic structures to construct a framework for the rapid and easy development of new, overloaded semantics for Python which dispatch directly to domain-specific libraries. This provides an easy mechanism for deep embeddings of DSLs in Python in the style of Rompf et al. [2012], rather than the shallow embedding offered by an API-centric approach. This strategy allows for tracing and translation as described above, but in a reusable, general-purpose setting. While this does eschew the extensible syntax provided by modifying the Python interpreter, we argue that the compatibility and portability benefits outweigh this limitation, as well as the benefit of avoiding altering the interface with which programmers are accustomed.

This paper makes the following contributions:

- We present a survey of existing techniques for developing new frameworks in Python, and motivate the use of virtualization in creating a general-purpose DSL framework (Section 2).
- We propose a structured methodology for performing source code transformations and virtualizations in Python. We show that this virtualization strategy allows for a more complete overloading of semantics in Python, as well as providing the ability to do multi-stage programming in Python (Section 3).
- We describe an implementation of this methodology in a system called Snek. Snek represents the first general-purpose, type-driven staging framework for a dynamic language (Section 4).
- We demonstrate using Snek to implement a DSL targeting the Z3Py interface. We show that Snek follows the proposed methodology, while allowing users to call Z3Py-specific calls where necessary, and idiomatic Python where possible (e.g., control flow). We further demonstrate how Snek provides the ability to perform multi-stage programming based on types (Section 5).
- We evaluate Snek by applying it to a wide variety of frameworks. We show that targeting a new framework using Snek takes minimal effort for DSL developers and that dispatching to that DSL requires a single decorator call for end users. We evaluate prototype implementations in Snek, translating code targeting three different numeric computing front-ends to instead target any of four different back-ends. We also show a more complete system which translates PyTorch to Lantern [Wang et al. 2018], implemented using the same methodology (Section 6).

While the paper focuses on Python exclusively, the core of the approach generalizes to any language; all that is required is a syntactic rewriting or pre-processing facility to replace uses of built-in primitives with late-bound function calls.

2 BACKGROUND

We begin by examining work which inspired the construction of a structured methodology for the virtualization of Python. We focus this discussion primarily on current techniques which aim to maintain the default Python interface while providing specialized runtime behavior.

2.1 Translation

A number of existing systems [Behnel et al. 2011; Lam et al. 2015; Palkar et al. 2018; PyTorch Contributors 2018; van Merrienboer et al. 2018] make use of source-to-source translation in order to bypass the overhead inherent in the Python interpreter [Al-Rfou et al. 2016] altogether. These operate on Python ASTs, which are then translated to a different, generally lower-level, language. We examine two systems which perform source-to-source translation targeting Python here, and provide a more exhaustive analysis in Section 7.

Cython. Perhaps the most well-known Python translation tool is Cython [Behnel et al. 2011]. Cython accepts as input a Python program, and generates from it an equivalent program using C as a "host" language. Cython is *not* a simple Python-to-C translation engine: it interfaces with the Python runtime so as to enable the use of Python objects should Cython be unable to generate the appropriate C code.

Given the Python program[1] in Figure 1 (a) as input, Cython will produce a .c file which runs approximately 35% faster than in Python [Bradshaw et al. 2011]. Notably, this file is just over 3300 LOC, with the majority of lines being constant definitions and other preprocessor directives (the original Python code is contained within a comment, but is otherwise difficult to find). Cython is able to generate even faster code if one supplies type annotations, as shown in Figure 1 (b).

```
def f(x): return x ** 2 - x
def integrate_f(a, b, N):
  s = 0
  dx = (b - a) / N
  for i in range(N):
    s += f(a + i * dx)
  return s * dx
```

(a)

```
def f(double x): return x ** 2 - x
def integrate_f(double a, double b, int N):
  cdef int i
  cdef double s, dx
  s = 0
  dx = (b - a) / N
  for i in range(N): s += f(a + i * dx)
  return s * dx
```

(b)

Fig. 1. Cython tutorial code without (left) and with (right) type annotations provided by users.

Running this annotated code with Cython yields slightly (~50 LOC) reduced code, but provides a speedup of approximately 4× over the original Python version [Bradshaw et al. 2011].

While these results are impressive, even in the trivial example shown, there exist additional optimization opportunities which are currently unavailable. If, for example, the value of `N` can become known at compile time (i.e., when Cython is invoked), Cython would not need to generate a `for` loop: rather, it could directly assign `s = f(a) + f(a + dx) + ... + f(a + (N - 1) * dx)`, thus removing the required jump operations inherent in the generated `for` loop.

TorchScript: **`torch.jit.script`.** Numeric computing libraries have also begun offering translation utilities. PyTorch's TorchScript [PyTorch Contributors 2018] framework provides such a translation mechanism: `torch.jit.script`. TorchScript's `torch.jit.script` function behaves similarly to Numba [Lam et al. 2015]: it takes as input a Python function which will be interpreted as a specialized subset of Python, in this case, TorchScript. Whereas tools like Cython typically require

[1] Taken from http://docs.cython.org/en/latest/src/quickstart/cythonize.html.

translation of entire programs (and, in the case of any errors, may require users to be proficient in the generated language), `torch.jit.script` instead allows users to mix these translations into their code where appropriate and with greater control (with errors appearing in a language similar to the original function). TorchScript is intended to be used for machine learning tasks, and provides the benefit of allowing users to more easily save machine learning models for later use in the form of a computation graph.

In order to achieve this flexibility, TorchScript (at the time of writing) imposes a number of limitations upon users. Of particular relevance here are the limitations that all functions decorated (Python decorators may be viewed as analogous to Java annotations) with `torch.jit.script` must return a value of type `tensor`, a function may only have a single return statement, and that control flow conditionals are only defined over `tensor` values. For example, the following code throws an error that `x` is a `Number`, rather than a `tensor` value:

```
@torch.jit.script
def foo():
  x = 3
  ret = None
  if x > 2: # currently unsupported in TorchScript
    ret = tensor.rand(1, 2)
  else: ret = tensor.rand(2, 3)
  return ret
```

Furthermore, although TorchScript in its current iteration does provide the benefit of increased usability for users, as with Cython, `torch.jit.script`'s current method of translation does not utilize any data *independent* information. Consider, for example, the function in Figure 2 (top). This produces a computation graph expressible as the control flow graph in Figure 2 (c). A decision point regarding `args.train` is present, despite the fact that this value will be static for the length of the program. Indeed, such a branching statement can and should be entirely removed. It should be noted that TorchScript already implements some dead code elimination through the use of liveness analysis, but opportunities such as this are currently overlooked.

Downside: no multi-stage programming. Cython, TorchScript, and similar systems which perform wholesale translation in this fashion are unable to utilize known data to specialize code generation and to take advantage of the ability to execute code *during* the translation. However, this is a well-studied technique known as *multi-stage programming* or *staging*, and existing work shows that this technique can be successfully implemented in higher-level languages [Rompf 2012; Rompf and Odersky 2010] (*in addition* to the translation techniques currently used by systems like Cython and TorchScript) and crucially allows to delegate modularity and abstraction to the meta-language, while keeping the DSL itself simple and easy for a compiler to optimize [Brown et al. 2011; Chafi et al. 2011].

2.2 Tracing

Rather than performing the wholesale translation described above, other systems elect to perform *tracing*: running operations and recording them in order to build up a representation of a program. We examine perhaps the most well-known machine learning framework which performs tracing in this fashion: PyTorch [PyTorch Contributors 2019].

TorchScript: **`torch.jit.trace`.** PyTorch [PyTorch Contributors 2019] performs tracing in the traditional fashion, though in order to provide other opportunities for optimization, PyTorch has also introduced a new method, `torch.jit.trace` (we refer to this as `trace` for the remainder of the paper), as part of the TorchScript framework. As with `script`, `trace` allows users to create models to be used at a later time, rather than, as with most tracing efforts, immediately upon completing the

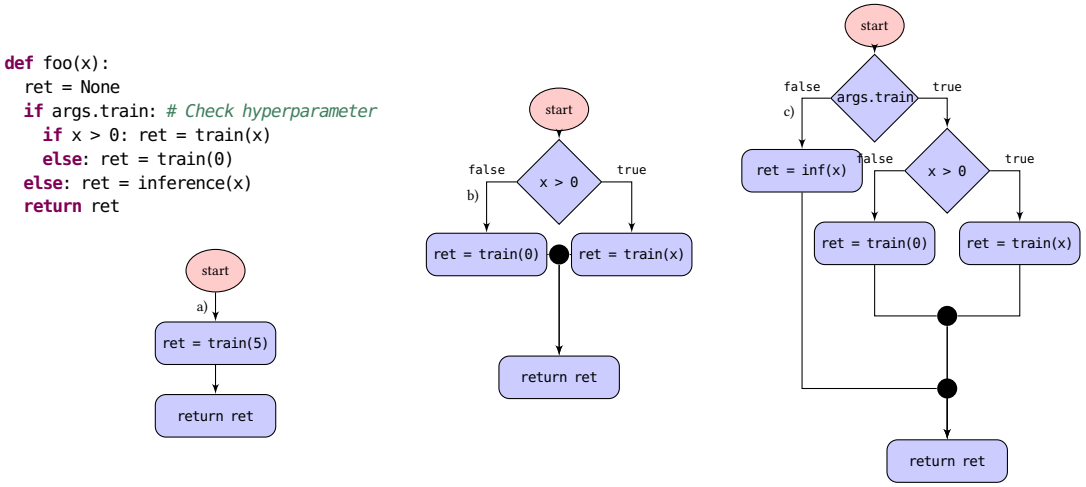

Fig. 2. Control Flow Graphs generated from function `foo` (top) using a) `torch.jit.trace` with a sample value of `x = 5` and `args.train` set to `True`, b) Snek with `args.train` set to `True`, and c) `torch.jit.script`.

trace. Tracing in this fashion is typically accomplished via operator overloading, thus requiring no additional effort on the part of the user, though `trace` does require users to provide sample input data for any traced functions.

Downside: no control flow. Consider the code in Figure 2 (top). Invoking `trace` on `foo` yields the control flow graph shown in Figure 2 (a). As with all CFGs produced via tracing, this graph is entirely linear; reusing a model generated in this fashion with a different value of `x` may produce invalid results. Thus, while tracing achieves the specialization benefits of multi-stage programming, it "overfits" and is unable to represent control-flow at the DSL level at all.

2.3 Virtualization

Chafi et al. [2010] proposed to "virtualize" built-in language features such as control-flow constructs by making them overloadable as virtual methods, much like operator overloading. In this virtualized form, language constructs yield the same high-level interface to which users are accustomed, but are also able to provide custom behavior. Crucially, this enables to extend the key idea behind tracing (just record operations) to control flow, thus removing its key limitation. Unfortunately, such virtualization is not immediately available in Python. Unlike, for example, the `__add__` method for overloading the `+` operator, there is no notion of overloading a magic `__if__` method, as it is not associated with a particular class.

In this paper, we propose to extend the operator overloading capabilities of Python to allow virtualization in the style of Rompf et al. [2012] through the use of source code transformations on the original program, effectively choosing to generate Python (rather than e.g., TorchScript) via preprocessing. In this generated, intermediate Python, we aim to produce constructs which framework developers may "overload" by providing custom implementations of the virtualized functions. Virtualization in this fashion allows for both tracing and translation as described in the manner above. Notably, this also allows for the optimization opportunities missed by translation efforts, while also capturing information which is lost by tracing efforts (see Figure 2 (b)). As shown by Moors et al. [2012], virtualization in this fashion also enables "staging based on types" exhibited by systems such as LMS (Lightweight Modular Staging) [Rompf and Odersky 2012]. We provide a

detailed description on how the principles from LMS can be applied in a dynamic language such as Python in Section 5.3.

AutoGraph. AutoGraph is a tool for TensorFlow that uses a limited form of virtualization to allow programmers to instantiate TensorFlow graphs using pure-python syntax [Moldovan et al. 2019]. Unlike TorchScript, AutoGraph allows programmers to mix Python semantics and TensorFlow semantics, dynamically deciding whether to stage an operation for later computation based on run-time type. In the example below `x > 2` is evaluated, if the result is a TensorFlow tensor (i.e. if `x` is a `Tensor`) then the conditional will be staged, if `x > 2` is a Python boolean then the code will be evaluated immediately.

```
@tf.autograph.convert()
def foo(x):
  ret = None
  if x > 2: # Staged or unstaged, based on the type of (x > 2)
    ret = 0
  else:
    ret = 1
  return ret
```

Dynamically deciding whether to stage an operation based on runtime type information is analogous to the static-type dispatch in systems like LMS. While the AutoGraph authors argue for the generality of this approach, the actual system and design choices were targeted toward making a specific back-end (TensorFlow) easier to program. For example, AutoGraph does not virtualize function calls, local variable access, attribute access, and other Python features that are not directly applicable. In addition AutoGraph's code generation is specific to TensorFlow. Their proof-of-concept example targeting the Lantern framework required rewriting most of the code generation machinery in AutoGraph, and is not present in the open-source version of AutoGraph. AutoGraph assumes a pure Python front-end targeting a single graph-based machine learning framework. Non-machine-learning applications were not considered. AutoGraph was not designed to target multiple frameworks within a single application, nor retarget from one framework to another.

Our approach, as implemented in Snek (introduced fully in Section 4), is a generalization of AutoGraph's, making it easy to override the functionality of Python in isolated modules. Adding a new back-end to Snek is essentially as easy as overloading an operator in Python. In addition, Snek virtualizes almost all Python features and allows developers to pick and choose the features required for their specific application. This extends, for example, to the type-based staging capabilities enabled by Snek.

3 CORE METHODOLOGY

We implement a generic methodology for performing virtualization of Python constructs. Our methodology makes it easy for a broad range of framework developers to override Python semantics with their own behavior, and does so in such a way that has minimal impact on the users of those frameworks. In order to achieve this latter goal, we establish the following criteria:

- The Python interpreter must remain unmodified. Providing a modified interpreter requires all existing Python instances to be replaced to achieve the proposed benefits, which is prohibitive for end users.
- As a byproduct of this first constraint, no new syntax may be added to the Python front-end.
- Following Language-Oriented Programming precepts, no extra-linguistic mechanisms should be required (i.e., this methodology should be implementable in the host language) [Felleisen et al. 2018].

$[\![p_1]\!]_v$, p_1 a top-level function	=	`def gen_func(overload):` `  `$[\![p_1]\!]_v$

$[\![$
`def` $f_1(p_1,...,p_n)$`:`
`  `s_1
$]\!]_v$
	=	`def` $f_1(p_x,...,p_y)$`:`
		`  `p_1 `= overload.init('`p_1`')`
		`  overload.assign(`p_1`, `p_x`)`
		`  ...`
		`  `p_n `= overload.init('`p_n`')`
		`  overload.assign(`p_n`, `p_y`)`
		
		`  `$[\![s_1]\!]_v$

$[\![$ `if cond:` s_1 `else:` s_2 $]\!]_v$	=	`def if_cond():` $[\![$ `cond` $]\!]_v$ `def if_body():` $[\![s_1]\!]_v$ `def if_orelse():` $[\![s_2]\!]_v$ `overload.if_stmt(if_cond, if_body, if_orelse, (local_writes,))`
$[\![$ `while cond:` s_1 `else:` s_2 $]\!]_v$	=	`def while_cond(): return` $[\![$`cond`$]\!]_v$ `def while_body():` $[\![s_1]\!]_v$ `def while_orelse():` $[\![s_2]\!]_v$ `overload.while_stmt(while_cond, while_body, while_orelse, (local_writes,))`
$[\![$ `for` s_1 `in` s_s`:` s_3 `else:` s_4 $]\!]_v$	=	`def for_body():` $[\![s_3]\!]_v$ `def for_else():` $[\![s_4]\!]_v$ `overload.for_stmt(`$[\![s_1]\!]_v$`,` $[\![s_2]\!]_v$ `for_body, for_else, (local_writes,))`
$[\![f_1(s_1,...,s_n)]\!]_v$	=	`overload.call(`$[\![f_1]\!]_v$`,` $[\![s_1]\!]_v$`,...,`$[\![s_n]\!]_v$`)`
$[\![$ `op` $s_1]\!]_v$	=	`overload.op_name(`$[\![s_1]\!]_v$`)` *`# e.g., not_`*
$[\![s_1$ `op` $s_2]\!]_v$	=	`overload.op_name(`$[\![s_1]\!]_v$`,` $[\![s_2]\!]_v$`)` *`# e.g., and_, or_`*
$[\![$ `x =` $e_1]\!]_v$	=	`x = overload.init('x')` *`# upon first assignment only`* `overload.assign(x,` $[\![e_1]\!]_v$`)`
$[\![$ `x`$]\!]_v$	=	`overload.read(x)`
$[\![$`_`$]\!]_v$	=	`_`

Fig. 3. Virtualization rules for Python constructs. We use fresh names for all extracted function names. All statements listed may represent multiple statements; we elide proper Python whitespace rules for presentation only.

- End users should be able to overload the semantics of individual functions independently, including the use of multiple semantic overload objects in a single application.

With these requirements in place, we must look to an implementation based in Python which modifies input programs at the source code level. Furthermore, all virtualizations which take place should happen at the granularity of functions: this enables a clear scoping boundary for transformation rules and framework developers (i.e., developers virtualizing source code using this methodology), as well as providing an easy implementation mechanism for end users (i.e., developers using the virtualized source code provided by the aforementioned framework developers).

3.1 Source Code Transformation and Virtualization Rules

We present the source code transformation rules required by our virtualization strategy in Figure 3. In these rules, `overload` represents an object upon which the corresponding method is defined (see Section 4) — an approach that has been described as tagless-final encoding [Carette et al. 2007], polymorphic embedding [Hofer et al. 2008], or object-algebra representation [d. S. Oliveira and Cook 2012; Guttag and Horning 1978] of program syntax. If `overload` contains functionality which mirrors the default Python semantics for the given construct, these virtualizations do not modify

the semantics of the original program (see Section 4.3 for a more formal examination). Furthermore, one could dynamically choose whether to virtualize the listed constructs: that is, if `overload` does not provide functionality for e.g., `if` statements, `if` statements need not be virtualized.

Control Flow. When virtualizing control flow structures, we extract the disparate computation pieces into corresponding functions (e.g., the conditional and each branch of an `if` statement), and create a call to `overload.if_stmt`. This virtualization is done as such for two primary purposes: first, this is consistent with frameworks which expect Python control flow to be expressed in a functional style, providing DSL developers with the intended structure; second, this is consistent with common compiler techniques such as conversion to continuation-passing style [Appel and Jim 1989]. Due to the fact that control flow structures are statements in Python, rather than expressions, there is no output to be captured (for a treatment on how `return` statements can be handled within control flow structures, see Section 5.3).

Variable Virtualization. In order to allow DSL developers control over all uses of a variable, including operations which are not by default overloadable in Python through "dunder" methods (e.g., `__add__`), we virtualize reads and assignments of variables. We virtualize all variables local to a function or which have been virtualized in an enclosing scope.

3.2 Virtualizing Function Calls

The virtualization of function calls is somewhat mechanical; the rules for doing so are provided in Figure 3. In essence, all function calls are wrapped in an `overload.call` (e.g., `print(x)` becomes `overload.call(print, x)`). This virtualization provides the powerful ability of replacing the runtime behavior of functions, even functions for which the source code may be unavailable or unknown, such as Python built-in functions (e.g., `len`, `range`).

To illustrate the benefits, consider the state-of-the-art numeric computing library, JAX [Frostig et al. 2018]. JAX serves as a "drop-in" replacement for NumPy [van der Walt et al. 2011]: users may take existing NumPy programs, rewrite the NumPy `import` statement (i.e., `import numpy as np` becomes `import jax.numpy as np`), and use the facilities provided by JAX such as just-in-time (JIT) compilation to specialized back-end hardware. In order to maintain the NumPy interface, the JAX developers were required to reimplement each user-facing function available in NumPy such that it executes JAX functionality in its place. Function call virtualization in the style described above allows for this logic to be separated from the JAX back-end; we can perform replacement at the time user code is run.

We note that JAX's JIT capabilities do not currently allow for idiomatic Python control flow. Indeed, the JAX developers have released documentation[2] which details the functional manner in which control flow must be constructed in order to avoid recompilation. As discussed in Section 6, however, applying the methodology as shown here provides users idiomatic control flow, while generating the corresponding functional-style control flow expected by JAX.

3.3 The Hard Parts

Scope. In perhaps the most comprehensive formal PL view of Python to date, Politz et al. [2013] demonstrate a number of features in Python's semantics which may appear counterintuitive to many users; of particular interest is Python's scoping mechanism. Python contains three types of variables in relation to scoping rules: `global`, `nonlocal`, and `local`. A simplified view is that all scopes have read-only access to all variables declared in any enclosing scopes (`nonlocal` variables), and write access to all variables declared in the current scope (`local`). `global` variables are stored in a

[2] https://github.com/google/jax/blob/master/notebooks/Common_Gotchas_in_JAX.ipynb

```
def g():
  x = 'not affected'
  def h():
    x = 'inner x'
    return x
  return (h(), x)

g() # ==> ('inner x', `not affected')
```

```
def g():
  x = 'not affected by h'
  def h():
    nonlocal x
    x = 'inner x'
    return x
  return (h(), x)

g() # ==> ('inner x', 'inner x')
```

Fig. 4. Example of nested function scopes in Python (left) and the effect of `nonlocal` (right). Originally appeared in Politz et al. [2013].

dictionary at the function level, which otherwise behave the same as `nonlocal` variables. For example, consider the code in Figure 4, (left). Here, we can examine an assignment to `x` in `h`, which defines a new variable (also named `x`), rather than updating the value of the outermost `x`. Using the `nonlocal` keyword, however, provides `h` with write access on `x` (Figure 4, right).

To view this in our proposed transformation strategy, consider the following program:

```
def assign_in_if(x, condition):
  if condition:
    x = x + 1
  else:
    x = x + 2
  return x
```

Transforming this code in this (naive) manner yields the following code:

```
def assign_in_if(x, condition):
  def if_cond(): return condition
  def if_body():
    x = x + 1
  def if_orelse():
    x = x + 2
  overload.if_stmt(if_cond, if_body, if_orelse)
  return x
```

Upon invocation of either `if_body` or `if_orelse`, this will yield an error due to Python's scoping rules. Attempting to read from this variable results in an error, as the variable is an *unbound local*.

This is resolved through the use of variable virtualization, as described above. Whereas all other virtualizations may take place without impacting the default Python semantics, control flow virtualization necessitates that variable virtualization has already occurred.

When this order is respected, the following `assign_in_if` is generated:

```
def assign_in_if(x_1, condition_1):
  x = overload.init('x')
  overload.assign(x, x_1)
  condition = overload.init('condition')
  overload.assign(condition, condition_1)
  def if_cond(): return overload.read(condition)
  def if_body():
    overload.assign(x, overload.read(x) + 1)
  def if_orelse():
    overload.assign(x, overload.read(x) + 2)
  overload.if_stmt(if_cond, if_body, if_orelse)
  return x
```

Due to these requirements, variable virtualization requires a separate pass to determine proper scopes.

A naive approach may be to use the `nonlocal` keyword, which allows modifications of variables declared outside the current block. However, such a solution is not viable for two primary reasons.

First, this prevents these transformations from being applied in earlier versions of Python (`nonlocal` was introduced in Python 3), as well as complicating the application of these rules to languages which do not have a `nonlocal` construct. Second, this does not allow for the virtualization of accesses and modifications for such variables.

Capturing `local_writes` in Control Flow Structures. It may be the case that *all* branches of a control flow structure need to be evaluated (e.g., in tracing) during the dispatch to DSL code. In such cases, the effects of this evaluation should be discarded, including writes to variables which take place. We note that such instances occur in structures where some paths may not be executed (e.g., control flow structures). Activity analysis, an analysis which crawls the AST and determines syntactically where variables may be accessed, may be used to accurately inventory these instances. This analysis can build a list of variables which may be written, which can then be passed to the corresponding DSL code (in Figure 3, this is represented as the `local_writes` parameter passed in the case of `if`, `while`, and `for`).

In essence, the proposed methodology virtualizes code blocks by transforming them into functions/ Due to Python's scoping mechanism, however, variables modified by the transformed block become local to the created function. This violates the original Python semantics, in which these variables must be treated as nonlocal variables (i.e., writes must affect variables in the original scope). Crucially, the virtualization of variable writes mitigates this problem, but staged code may not have a corresponding mechanism (e.g., staged control flow may use value semantics). As such, this set of `local_writes` is required to retroactively supply the modified symbols.

Resolving Closures for `overload`. As shown in the first rule of Figure 3, upon applying our methodology to a program, the program must be wrapped in a generated function which takes as parameter the `overload` object containing the relevant DSL code. However, we must capture all "metadata" relevant to the program p_1 (e.g., `global` or closure variables) and attach this to the newly generated function, `gen_func`. Python tracks this information at the *code object* level for functions, which provides an easy interface for our purposes.

Return Statements. Virtualizing control flow in the manner described poses some difficulty when `return` statements appear in branches. For example, consider the following code:

```
def abs(x):
  if x < 0:
    return -x
  else:
    return x
```

Transforming this code using the `if` virtualization rules in Figure 3 yields the following (we elide variable virtualization for presentation):

```
def abs(x):
  def if_cond():
    return x < 0
  def if_body():
    return -x
  def if_orelse():
    return x
  overload.if_stmt(if_cond, if_body, if_orelse)
```

In this function, the value returned by `overload.if_stmt` will be ignored and lost, causing `abs` to always return `None`, rather than the intended value. This problem may be rectified in a number of ways; we do not enforce one option over the other in our proposed methodology. One such solution may be found in AutoGraph, which elects to track all values found in a `return` statement, and return these values at the end of a function. Another solution may be to wrap the contents of generated functions in a `try/except` block, and upon a virtualized function reaching a `return` statement, raise

an `Exception` which contains the return value (this is described further in Section 6.2). We note that both such strategies are expressible using the methodology as thus proposed.

Other forms of nonlocal control flow (e.g., `break`, `continue`) incur a similar problem. For `break`, one may modify the containing loops such that they are conditioned on an additional `boolean` flag; this flag represents whether `break` has been called, and may be set in place of the original `break` statement. `continue` may be handled in a similar fashion. Another possibility for both `break` and `continue` is to use an `Exception`-based mechanism which, upon reaching the statement, throws the corresponding `Exception`. The appropriate action can then be taken in the handler of the `Exception`, provided in the implementation of the virtualized function. Similar behavior can be implemented to handle `raise` statements.

4 SNEK

In this section, we present an implementation of the proposed methodology as a system called Snek. Snek works by parsing a Python function, walking its AST, and applying the corresponding virtualization transformations as described in Figure 3. Users of Snek then provide an object containing implementations of the virtualized functions to be invoked at run time in place of the default Python semantics. Snek is agnostic to the type of this object: for presentation purposes, we will assume this object represents a module containing the relevant functionality, though an object-oriented approach based on Python `classes` is also possible.

In this section, we refer to users of Snek as "DSL developers", as they provide domain-specific logic which uses Python as a front-end. While a slight abuse in terminology (as these users are not defining new syntax), Snek allows users to view Python as a DSL for which they provide a custom interpretation.

4.1 Virtualizing Using Snek

Snek provides as its primary interface `convert(func, overload)`, where `func` represents the function to be converted and `overload` is the object containing the virtualized implementations for a subset of transformations from Figure 3. In this manner, DSL developers are able to determine which features are actually virtualized using the `overload` object. For example, a DSL developer may wish to virtualize `while` loops but not `if` statements; in this case, they must simply provide an `overload` object which does not have `if_stmt` defined, but does have `while_stmt` defined.

4.2 Default Implementations

As discussed in Section 3.3, virtualizing control flow requires that variable virtualizations have taken place in order to have a correct virtualized program. However, it is undesirable to require DSL developers to implement behavior for variable virtualization when the default Python behavior exhibited in the original is the desired behavior. Furthermore, it can be difficult to accurately model this behavior, with a number of possible implementations, with varying levels of complexity.

To accommodate situations like this, Snek provides a `py_defaults` module which contains default implementations for all virtualized constructs. These default implementations can then be used in semantic overloads objects provided by DSL developers (e.g., `assign = py_defaults.assign`).

As an example, we show the `py_defaults.if_stmt` implementation, below.

```
def if_stmt(cond, body, orelse, _):
  if cond():
    body()
  else:
    orelse()
```

4.3 Towards Proving Correctness

λ_π as presented by Politz et al. [2013] is an executable small-step operational semantics written in PLT Redex [Felleisen et al. 2009] for Python[3], with an accompanying interpreter implemented in Racket. $\lambda_{\pi_\downarrow}$ is also provided in the current implementation[4], which serves as a set of desugaring rules capable of transforming any Python program into its core syntax.

As discussed in Section 3.3, Python's scoping rules, in particular, cause difficulty in performing transformations on Python code, requiring some form of variable lifting in order to correctly capture the intended Python semantics. λ_π introduces a special value, ☠, which is used to represent uninitialized heap locations. All identifier declarations are lifted to the top of their enclosing scope and given an initial value of ☠: if this value is ever read, it signals the use of an uninitialized identifier. λ_π provides a desugaring of `nonlocal` and `global` scopes and keywords, which serves to fully capture the scoping semantics of Python.

In order to formally examine our virtualization transformation $[\![\ ]\!]_v$, we implement the rules in Figure 3 in the form of reduction semantics. We accomplish this by adopting the reduction semantics presented in λ_π, and formulating our semantic preservation property in conformance thereof. The general form of the reduction relation ($\rightarrow$) is a pair of triples $(e, \varepsilon, \Sigma) \rightarrow (e, \varepsilon, \Sigma)$ where e are expressions, ε are global environments, and Σ are heaps. We denote the multiple step relation as $\rightarrow^*$.

We begin by introducing `dom`, a return set of variable references given a heap object: $\Sigma \rightarrow \mathbb{P}(\text{ref})$. We also introduce two auxiliary functions which capture the side effects introduced in $[\![\ ]\!]_v$. Given an expression, the first function $MV : e \rightarrow \mathbb{P}(\texttt{ref})$ returns the existing variable references *modified* by our transformation:

$MV(x = e) = \{x\},\ MV(deff\ldots) = \{f\},\ MV(\_) = \{\}$

The second function $NV : e \rightarrow \mathbb{P}(\texttt{ref})$ returns the variable references *created* by our transformations:

$NV(if\ldots) = \{fresh(then), fresh(else)\}$
$NV(while\ldots) = \{fresh(body), fresh(cond)\}$
$NV(for\ldots) = \{fresh(body)\}$
$NV(\_) = \{\}$

Definition ($\simeq_v$): given a well-formed Python program e, $e \simeq_v [\![e]\!]_v$ iff

(1) e diverges and $[\![e]\!]_v$ diverges, or
(2) e is stuck and $[\![e]\!]_v$ is stuck, or
(3) starting from ε and Σ, there exists some value v and heaps such that $(e, \varepsilon, \Sigma) \rightarrow^* (v, \varepsilon', \Sigma' \cup \Sigma_{MV})$ and $([\![e]\!]_v, \varepsilon, \Sigma) \rightarrow^* (v, \varepsilon', \Sigma' \cup \Sigma_{MV^*} \cup \Sigma_{NV})$, $\texttt{dom}(\Sigma') \cap \texttt{dom}(\Sigma_{MV}) = \varnothing$, $\texttt{dom}(\Sigma_{MV}) = \texttt{dom}(\Sigma_{MV^*}) = MV(e)$, $\texttt{dom}(\Sigma') \cap \texttt{dom}(\Sigma_{MV^*}) \cap \texttt{dom}(\Sigma_{NV}) = \varnothing$, and $\texttt{dom}(\Sigma_{NV}) = NV(e)$.

The third case specifies the behavior after transformation: First, Σ', the variable references not contained in $MV(e) \cup NV(e)$ remain untouched, and our transformation preserves the effects on that part. Second, the variable references in $MV(e)$ will be updated to the new heap Σ_{MV^*}. Third, the variable references in $NV(e)$ exist in the new heap Σ_{NV}, but *not* in the one before transformation. And lastly, these heaps are disjoint (i.e., there is no overlap in the respective domains).

Proposition: $\simeq_v$ is a congruence. If e is a well-formed Python program and $e \simeq_v [\![e]\!]_v$, then for any evaluation context E, we have $E[e] \simeq_v E[[\![e]\!]_v]$. As an example of this, we provide $[\![\ ]\!]_v$ for `if` statements expressed as a reduction rule implemented in PLT Redex (Figure 5).

We note that λ_π does not provide a proof of correctness. Rather, it provides an extensive suite of test cases which serve to provide strong evidence that an implementation in PLT Redex functions

[3]Python version 3.2.3
[4]https://github.com/brownplt/lambda-py

```
(→ ((in-hole E (if e_1 e_2 e_3)) ε Σ)
   ((in-hole E
     (let (thn-f local = (fun () (no-var) e_2)) in
       (let (els-f local = (fun () (no-var) e_3)) in
         (app (fun (test thn els)
                (no-var)
                (if (id test local)
                     (return (app (id thn local) ()))
                     (return (app (id els local) ()))))
           (e_1
            (id thn-f local)
            (id els-f local)))))) ε Σ)
   (where thn-f (gensym 'then))
   (where els-f (gensym 'else))
   "E-VirtIf")
```

Fig. 5. PLT Redex implementation of $[\![\ ]\!]_v$ applied to a Python `if/else` statement in λ_π.

in a manner equivalent to the default Python semantics. Furthermore, this evidence as it applies to Snek provides information only about $[\![\ ]\!]_v$ and the implementations supplied in the `py_defaults` module; no other functionality in Snek or modules provided by DSL developers have been compared against λ_π in our current work. While this is a step toward a formal guarantee of correctness, such a guarantee remains the subject for future work.

4.4 Current Limitations

Snek does not currently virtualize all Python constructs, though expanding coverage is simply an engineering effort: as shown by the λ_π work [Politz et al. 2013], there exist alternate implementations of all Python constructs, assuming one constructs a correct model for Python's scoping. For completeness, we describe the constructs which are not yet supported by Snek, as well as the appropriate implementation details necessary to support them.

4.4.1 Generators. Generators in Python may be virtualized in the same manner as `return` statements (see Section 3.3), with an auxiliary data structure which models the stack such that repeated execution of the statement yields the appropriate value. We note that implementing this in Snek is possible without modifying the current implementation strategy: generator expressions exist as a single AST node.

4.4.2 Exceptions. As discussed in Section 3.1, many of the transformation rules shown follow standard continuation-passing style. It is thus trivial to provide an additional argument to generated functions representing the exception handler(s) associated with the original, virtualized code block. This would allow Snek to execute the generated function within a `try/except`, in which caught exceptions are delegated to their appropriate (virtualized) handlers.

4.4.3 Classes. Section 4.3 discusses how λ_π [Politz et al. 2013] desugars all Python constructs into its core, PLT Redex-based, syntax. This includes the desugaring of classes into collections of functions and properties, internally represented as a new type. This same approach can be taken by Snek, and, indeed, has been shown in the implementation of AutoGraph [Moldovan et al. 2019].

4.4.4 Context Managers. Context managers in Python consist of `with` statements, of the structure:

```
with X(A) as Y:
    S
```

In general, such statements can be desugared as follows:

```
X1 = X(A)
Y = X1.__enter__()
S
```

```
def and_(p, qs):
  ret = z3.And(p, qs[0])
  if len(qs) > 1:
    for q in qs[1:]:
      ret = z3.And(ret, q)
  return ret
```
(a)

```
def or_(p, qs):
  ret = z3.Or(p, qs[0])
  if len(qs) > 1:
    for q in qs[1:]:
      ret = z3.Or(ret, q)
  return ret
```
(b)

```
def not_(p):
  return z3.Not(p)
```
(c)

Fig. 6. Implementations of `and` (a), `or` (b), and `not` (c) for Z3Py.

```
X1.__exit__()
```

This can then be virtualized following the rules shown in Section 3.

4.4.5 `async`. Asynchronous statements such as `async` and `await` can be trivially virtualized as function calls. Correctly modeling these in the default implementation provided by Snek may be done via the existing Python `asyncio` module, though such statements would require special handling due to their effectful nature (see Section 5.2 for a more complete discussion).

4.4.6 Native Bindings. Similar to asynchronous statements, native binding can be trivially transformed via function call virtualization, and a default implementation may rely on the standard modules provided by Python.

4.4.7 Staged Attribute Mutation. While Snek does not currently support staged attribute mutation (e.g., data-dependent `__getattr__` and `__setattr__` methods), the virtualization of these methods follows function call virtualization as shown in Figure 3. Providing a semantics-preserving default implementation may be done by maintaining a virtual function table and utilizing the existing function call virtualization capabilities.

4.4.8 Functions Without Source Code Information. Passing an object to a function into which Snek has no visibility (i.e., variable virtualization as described in Section 3.3 has not been performed) may result in an inconsistent state with respect to the original Python semantics if that object is modified. It is possible to mitigate this in nearly all cases through the use of virtualized `__getattr__` and `__setattr__` methods which, upon any modifications being performed to the object, dispatch to the appropriate virtualized code. While there exist cases such that this is insufficient (e.g., the use of `object.__setattr__` in place of the assignment operator), this limitation comes as a result of the myriad interfaces supplied by Python to accomplish similar tasks. We note that Snek provides support for what are currently considered the acceptable (i.e., "Pythonic") methods, and virtualization in the proposed style may not be limited in the same fashion when targeting a different language.

We note that `eval` and `exec` are also currently unsupported in Snek, though virtualization of these follows the function call virtualization rules described in Figure 3.

5 DESIGNING DSLS USING SNEK: Z3PY

In this section, we look at designing a new DSL using Snek to target the Z3Py [Microsoft Research 2019] interface. Z3Py is the Python front-end for the high performance theorem prover, Z3 [de Moura and Bjørner 2008].

5.1 De Morgan's Laws: `and`, `or`, and `not`

Consider the following definition of one of De Morgan's laws[5] written in Z3Py:

```
def z3_demorgan(p, q):
  return z3.And(p, q) == z3.Not(z3.Or(z3.Not(p), z3.Not(q)))
```

[5] Adapted from https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

```
queens = [z3.Int('queens_%i' % (i + 1))
            for i in range(8)]
ranks = [z3.And(1 <= queens[i], queens[i] <= 8)
            for i in range(8)]
files = [z3.Distinct(queens)]
diagonals = [z3.If(i == j,
               True,
               z3.And(queens[i] - queens[j] != i - j,
                 queens[i] - queens[j] != j - i))
             for i in range(8) for j in range(i)]
```

(a)

```
queens = [z3.Int('queens_%i' % (i + 1))
            for i in range(8)]
ranks = [1 <= queens[i] and queens[i] <= 8
            for i in range(8)]
files = [z3.Distinct(queens)]
diagonals = []
for i in range(8):
  for j in range(i):
    if i == j:
      diagonals.append(True)
    else:
      diagonals.append(queens[i] - queens[j] != i - j \
        and queens[i] - queens[j] != j - 1)
```

(b)

Fig. 7. Naive implementation of Eight Queens problem targeting Z3Py using the Z3 interface (a), compared with a naive implementation in idiomatic Python using Snek (b).

We can then use Z3 to prove the validity of this function, as follows:

```
def prove(f):
  solver = z3.Solver()
  solver.add(z3.Not(f))
  return solver.check() == z3.unsat

p, q = z3.Bools('p q')
assert prove(z3_demorgan(p, q))
```

Examining the code in `z3_demorgan`, one may notice that all calls to Z3Py simply mirror constructs already present in Python: `and`, `not`, and `or`. As such, one may imagine rewriting this function using only Python constructs:

```
def demorgan(p, q):
  return (p and q) == (not (not p or not q))
```

However, invoking this function using a `z3.Bool` value fails, as this would attempt to directly evaluate the symbolic expressions, rather than build a Z3 representation as in `z3_demorgan`.

Snek provides a `convert` method which takes as parameters the function to be converted and the `overloads` object containing the implementations of the virtualized constructs. We can thus provide the implementations of `and_`, `or_`, and `not_` as shown in Figure 6[6] in a `z3py_overloads` object. We can then convert `demorgan` and see the same behavior as with `z3_demorgan`, as follows:

```
converted_demorgan = Snek.convert(demorgan, z3py_overloads)
p, q = z3.Bools('p q')
assert prove(converted_demorgan(p, q))
```

5.2 Eight Queens: Control Flow and Isolated Execution

While the constructs demonstrated thus far are sufficient for linear programs in Z3, Z3 also contains API calls for control flow. Z3 allows users to construct `if` statements of the following form: `z3.If(a, b, c)` (Python equivalent: `b if a else c`). This is a useful abstraction, but becomes cumbersome when nesting control flow. Consider, for example, the code in Figure 7 (a)[7]. Here, we must satisfy three constraints: all queens must be on a valid rank (`ranks`), no queens may share a file (`files`), and no queen may share a diagonal (`diagonals`). In constructing the constraint on diagonals, users must provide a somewhat unwieldy set of conditions, including a "then-branch" which adds a useless constraint of `True` due to the signature of `z3.If`. We note that this expression can actually be

[6]Note that in order to allow patterns such as `a and b and c`, Python constructs `and` and `or` operators as an operation on the first operand, with all other operands in a list as the second operand.

[7]Adapted from https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

greatly simplified, as `i == j` here will always evaluate to `false`. We avoid such simplifications in our presentation to stay true to the original presentation as possible, though it is arguable that such trivial simplifications may be found more easily in idiomatic Python.

Using Snek, we can replace this expression using an idiomatic Python `if` statement, as well as idiomatic Python logical operators (Figure 7 (b)). With a vision of the code we wish to write, as well as a knowledge of the code we wish to generate (Figure 7 (a)), we look to implementing `if_stmt`. However, the implementation of `if_stmt` is not as straightforward as those shown in Section 5.1. Consider the following naive approach:

```
def if_stmt(cond, body, orelse):
  return z3.If(cond(), body(), orelse())
```

`z3.If` requires Z3 *values* (i.e., not functions) for both the "then" and "else" branches, requiring our implementation to evaluate both branches. Doing so, however, will cause both calls to `diagonals.append` to take place, regardless of the result of `i == j`. Z3 does not allow, e.g., lambda functions to be passed as parameters to `z3.If`: we must have some method of evaluating the branches and discarding side effects. To accommodate this complication with control flow, Snek provides an `execute_isolated` function, which takes as parameters a nullary function representing the control flow path to be executed, and a list of free variables in that function which may be modified:

```
def execute_isolated(func, func_freevars):
  original_vals = [var.val for var in func_freevars]
  return_vals = func()
  modified_vals = [var.val for var in func_freevars]
  for var, val in zip(func_freevars, original_vals):
    var.val = val
  return modified_vals, return_vals
```

Note that while this function does protect against side effects which result from variable mutation, it does not protect against side effects which may result from I/O operations. It is possible to preemptively protect against such operations (e.g., one can imagine virtualizing these operations and preventing them from occurring while in `execute_isolated`). Snek provides the mechanism to virtualize said operations but the specific implementation must be provided by the DSL developer.

Utilizing `execute_isolated`, we can update our `if_stmt` as follows:

```
def if_stmt(cond, body, orelse, local_writes):
  body_vals, _ = execute_isolated(body, local_writes)
  orelse_vals, _ = execute_isolated(orelse, local_writes)

  body_result, else_result, modified_var in zip(body_vals, orelse_vals, local_writes):
    modified_var.val = z3.If(cond(), body_result, else_result)
```

With this in place, we can successfully convert the implementation in Figure 7 (b) and achieve the expected behavior.

5.3 Staging

Multi-stage programming allows programs to be viewed in multiple execution phases, with earlier phases informing the code generation (and specialization) of later phases. Of most relevance to Snek is the Lightweight Modular Staging (LMS) [Rompf and Odersky 2010] framework, which performs "staging based on types". Programmers use a provided type constructor to explicit delineate between information known at compile-time and that known at run time. LMS targets a Scala front-end, and utilizes the fact that Scala is statically typed in order to allow this type-based differentiation of stages.

Snek likewise separates programs into explicit execution phases: code as written by the user, which is virtualized and rewritten; and the execution of that code. It is during this execution stage that the code contained within an `overload` object is executed, rather than the code as originally

```python
def and_(a, b):
  if isinstance(a, z3.BoolRef):
    if len(b) > 1:
      return_val = and_(b[0](), b[1:])
    else:
      return_val = b[0]()

    if isinstance(return_val, z3.BoolRef):
      return z3.And(a, return_val)
    else:
      if return_val:
        return a
      else:
        return False
  else:
    if a:
      if len(b) > 1:
        return and_(b[0](), b[1:])
      else:
        return b[0]()
    else:
      return False
```

(a)

```python
def or_(a, b):
  if isinstance(a, z3.BoolRef):
    if len(b) > 1:
      return_val = or_(b[0](), b[1:])
    else:
      return_val = b[0]()

    if isinstance(return_val, z3.BoolRef):
      return z3.Or(a, return_val)
    else:
      if return_val:
        return True
      else:
        return a
  else:
    if a:
      return True
    else:
      if len(b) > 1:
        return or_(b[0](), b[1:])
      else:
        return b[0]()
```

(b)

Fig. 8. Staged implementations of `and` (a) and `or` (b) for Z3Py.

```python
def foo(c):
  x = 1

  if c:
    x = 2
  else:
    x = 3

  print(x)
```

(a)

```
def foo(c: Rep[Boolean]) = {
  var x = 1;

  if (c) {
      x = 2
  } else {
      x = 3
  }

  System.out.println(x)  // Unstaged print: undefined behavior
  print(x)               // Staged print
}
```

(b)

Fig. 9. Example showing undefined behavior in a) Python (Snek) and b) Scala (LMS), when printing.

written. Due to the fact that this code will be run on live objects, rather than on static code (as in the case of the rewriting phase), type information is exposed to the DSL developer. This enables "staging based on types" as presented in LMS, by directly dispatching based on the given dynamic type and/or generating code.

Z3Py provides a *deferred API*: constructing a Z3Py object (e.g., `z3.And`) simply constructs an object to be evaluated later, rather than performing any work in the call. This inherently breaks computation phases into stages: Z3 expression construction phase, and a later phase in which the Z3 runtime is invoked.

Consider the Z3Py implementations of `and_`, `or_`, and `not_` as shown in Figure 6. The functions `z3.And`, `z3.Or`, and `z3.Not` are able to accept Python `bool` objects as well as `z3.BoolRef` objects, but do not perform any kind of specialization or optimization based on the parameters (e.g., if any parameter in a `z3.And` expression is known to be `False`, the expression may be replaced with `False`). Attempting to add these optimizations would traditionally require modifying the Z3 source code. However, by providing an intermediate stage (in which the virtualized functions are evaluated), we can add these

optimizations *without* modifying the Z3Py implementation. We provide implementations in Figure 8 of `and_` and `or_` which bypass the Z3 runtime altogether where possible (a staged implementation of `not_` follows mechanically).

Difficulties in Staging. While staging as thus described seems straightforward, there exist some difficulties which may not be obvious at first glance. For example, consider the code in Figure 9 (a). Here, we must determine how to treat `x` if `c` is staged. LMS enforces a policy that all mutable variables are, by default, staged (i.e., `x` is lifted as a `Rep[Int]` in Figure 9 (b)). This is due to the fact that in many cases, it is likely the desire that side effects (e.g., printing) appear at run time, rather than at staging time. It is possible in LMS to override this default behavior by supplying an explicit type annotation, but doing so introduces undefined (and likely undesired) behavior.

Due to Python's dynamic type system, DSL developers may choose to modify the type of `x` upon staging the `if` statement and replace it with a staged type. This, however, will cause the `print` statement to execute on a staged value: without `__repr__` defined on this staged type, this may cause an error. As such, it becomes necessary to provide a method whereby we may safely print (potentially) staged values. A naive solution may be to simply virtualize the `print` function in Python such that it dispatches based on the type of the argument. Snek provides a `RewritingCallOverload` utility which allows DSL developers to decorate implementations with `call.replaces(f)`, where `f` represents the front-end function which is being replaced. For example, virtualizing `print` to dispatch based on type of the argument may look as follows;

```
call = RewritingCallOverload(py_defaults.call)
@call.replaces(print)
def my_print(s):
  if should_stage(s):
    staged_print(s)
  else:
    print(s)
```

However, this implementation may cause unintended behavior: it is now impossible to print any current stage (i.e., known) values at run time! Due to the fact that Python's print statement allows an arbitrary number of parameters, we may adopt the solution proposed by Amin and Rompf [2018] and require that users provide a file descriptor as the first argument to `print` which determines at what point the value is output (i.e., whether the `print` statement executes or generates code). Such a technique will produce the intended behavior, but note that this requires modifying user code.

Another method may be to provide an explicit staged printing function. This follows the pattern of LMS (see Figure 9 (b)). This provides a natural solution, but does require users to modify existing code and to know whether a value is staged.

Yet another potential solution may be to provide a new function which returns a staged version of the given value (i.e., `T` ⇒ `Rep[T]` in LMS). Use of this function may be as follows:

```
print(rep('this will be staged!'))
```

Again, this requires modification of the original user code, and likewise requires the user to know whether a value may be staged. It does, however, provide the benefit that the DSL developer is able to determine the value returned by `rep`.

We note that any of these patterns is possible in Snek. Indeed, a DSL developer may elect to override the `__repr__` method of a staged type and simply print all values at staging time, with a custom message informing the user that a value is not yet known.

5.4 Simplifying Eight Queens with Virtualized Function Calls

Consider the implementation of the Eight Queens problem as shown in Figure 7 (b). While this implementation makes use of idiomatic Python control flow, the `if` statement which encodes the

```
queens = [z3.Int('queens_%i' % (i + 1))
            for i in range(8)]
ranks = [1 <= queens[i] and queens[i] <= 8
            for i in range(8)]
files = [z3.Distinct(queens)]
diagonals = []
for i in range(8):
  for j in range(i):
    if i != j:
      diagonals.append(queens[i] - queens[j] != i - j \
        and queens[i] - queens[j] != j - i)
```

(a)

```
queens = [z3.Int('queens_%i' % (i + 1))
            for i in range(8)]
ranks = [1 <= queens[i] and queens[i] <= 8
            for i in range(8)]
files = [z3.Distinct(queens)]
diagonals = []
for i in range(8):
  for j in range(i):
    if i != j:
      diagonals.append( \
        abs(queens[i] - queens[j]) != abs(i - j))
```

(b)

Fig. 10. Simplifying the Z3Py implementation.

Table 1. Results of solving the Eight Queens problem using the Z3Py interface. Results reported as the average of 50 runs, in seconds.

	Figure 7 (a)	Figure 7 (b)	Figure 10 (a)	Figure 10 (b)
Eight Queens	0.060	0.060	0.059	0.068

constraint that no two queens reside on the same diagonal retains the undesirable code of adding multiple (useless) `True` constraints. This can be rectified trivially by simply negating the condition and removing the corresponding `then` branch, as shown in Figure 10 (a).
The astute reader will note that this can be simplified further, as follows:

```
if i != j:
  diagonals.append(abs(queens[i] - queens[j]) != abs(i - j))
```

Without function call virtualization, this code results in an error at run time: `z3.ArithRef` (the type of `queens[i] - queens[j]`) has no `__abs__` method defined, causing `abs(queens[i] - queens[j])` to fail. Whereas traditionally implementing this method would require a developer to modify the Z3 codebase, we may provide the desired `abs` functionality at run time using the replacement utility described in Section 5.3:

```
@call.replaces(abs)
def z3_abs(x):
  if isinstance(x, z3.ArithRef):
    return z3.If(x < 0, -x, x)
  else:
    return abs(x)
```

With the ability to use `abs` on `z3.ArithRef` objects, we can reimplement the naive implementation in Figure 7 (a) with a simplified version written using native Python control flow and Python built-in functions, as shown in Figure 10 (b).

5.5 Performance

We show the results of running the Eight Queens implementations shown in Figures 7 and 10 in Table 1. Unsurprisingly, the performance of the Z3Py implementation (Figure 7 (a)) match exactly the performance of the naive Snek implementation (Figure 7 (b)), as the naive Snek implementation generates the exact same Z3 representation. Figure 10 (a) produces slightly better performance, as the generated Z3 representation is comprised entirely of `z3.And` expressions, rather than the `z3.If(False, True, ...)` pattern present in the representation generated by the naive implementations (Figure 7).

The simplified implementation presented in Figure 10 (b), on the other hand, runs more than 10% slower than the naive implementations. This is due to our implementation of `z3_abs`: it generates `z3.If`

Table 2. RNN Cell Performance (examples/sec, higher is faster)

Front-ends \ *Back-ends*	PyTorch	TF Eager	NumPy	TF Graph
PyTorch	29.46	84.09	29.73	100.37
TF Eager	29.25	86.94	29.38	105.70
NumPy	29.42	80.30	29.54	109.78
TF Graph	-	-	-	101.67

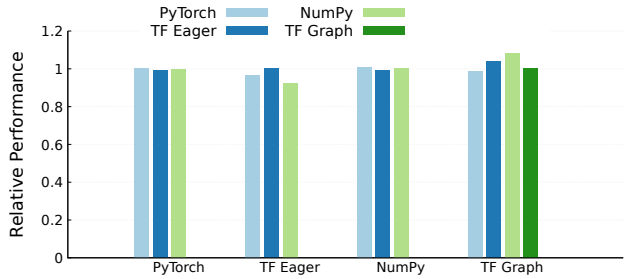

Fig. 11. Relative performance for a dynamic RNN using Snek on PyTorch, TensorFlow Eager (TF Eager), and NumPy as front-ends, with PyTorch, TensorFlow Eager, NumPy, and TensorFlow Graph as back-ends in comparison to an unmodified front-end running on the same back-end (e.g., PyTorch code running on PyTorch). Note that TensorFlow Graph has not been implemented as a front-end.

nodes which replace the `z3.And` nodes found in Figure 10 (a). Examining this node provides some clarification: `z3.If((queens[i] - queens[j]) < 0, -(queens[i] - queens[j]), (queens[i] - queens[j]))`. While Z3 may perform the equivalent of common subexpression elimination to avoid computing `queens[i] - queens[j]` multiple times, this is still additional overhead which the other provided implementations are able to avoid.

5.6 Applying Snek

To this point, we have examined using Snek as a DSL developer. The next step as such a developer is to provide an interface to apply the conversion. As we do not wish to expose the `convert` function provided by Snek, we may instead build a Python decorator, as follows:

```
def z3(func):
  return Snek.convert(func, z3py_overloads)
```

Making this available to end users allows them to apply this, as follows:

```
@z3
def demorgan(p, q):
  return (p and q) == (not (not p or not q))
```

No additional changes are required from users, as this decorator will apply the modification in-place (i.e., all calls to `demorgan` will call the converted function).

6 EVALUATION

To demonstrate the flexibility of Snek we use it to define translations between several popular numerical computing frameworks. This also allows us to quantify some of the performance effects of virtualization. Each of these frameworks has a unique Python API, some even have different execution semantics (e.g. PyTorch's define-by-run semantics vs. TensorFlow's define-then-run

semantics). For this evaluation, we do not attempt to translate the entire API, but rather only the subset needed to demonstrate proof of concept. We do virtualize control flow for all examples.

We also describe a more complete example, using Snek to generate code targeting the Lantern [Wang et al. 2018] back-end. This example takes code written for PyTorch and translates it to target Lantern. We demonstrate the effectiveness of this translation on a standard MNIST benchmark.

Nearly all experiments were conducted on a single machine with a dual-threaded 6-core Intel Xeon E5- 1650 CPU. The one exception is the MNIST experiment, which was conducted on a single NUMA machine with 4 sockets, 24 Intel Xeon Platinum 8168 CPUs per socket, and 750 GB of RAM per socket. We used Python 2.7.1[8], GCC 5.4.0, TensorFlow 1.13.1, PyTorch 0.4.0, and Scala 2.11.6.

6.1 Dynamic RNN

We used Snek to retarget code from one numerical computing framework to another. As a running example we implemented a simple recurrent neural network model in PyTorch, NumPy, and TensorFlow's Eager mode. Each of these is targeted at several back-ends: PyTorch, NumPy, TensorFlow Eager, and TensorFlow Graph. We report the performance of running the original and translated code in Table 2. We also include a handwritten TensorFlow graph version for comparison. Note that we do not translate this graph version; this is not a typical use case due to TensorFlow Graph's non-intuitive programming model. Each entry in the table is the average of 50 runs each with 50 batches of 32 randomly generated examples. Each example has a maximum sequence length of 64.

By reading a column of Table 2 we can compare the performance of untranslated code (where the front-end and back-end are the same) with code targeted from a different framework. As can be seen from Figure 11, for each back-end, performance does not depend on whether the code was translated from a different front-end or written by hand.

6.1.1 PyTorch. The PyTorch implementation of a minimal RNN model can be seen below.

```
def rnn_model(inputs, seq_len, w, b, init_state):
  """Very basic RNN model, implemented in PyTorch."""
  inputs_time_major = torch.transpose(inputs, 0, 1)
  max_seq_len = torch.max(seq_len)
  state = init_state
  for i in torch.arange(max_seq_len):
    x = inputs_time_major[i]
    h = torch.cat((x, state))
    state = torch.nn.tanh(torch.mm(h, w) + b)  # Basic RNN cell
  return state
```

We can see that this implementation, while minimal, has some features that would be challenging in a syntactic approach to translation. First of all there are a number of calls into the PyTorch API each of which has to be mapped to corresponding logic in the other frameworks. Second there is data dependent control-flow. The `for` loop depends on a value `max_seq_len` that is computed at run time. For define-by-run frameworks like PyTorch and TensorFlow Eager the control flow is executed within the Python runtime. For define-then-run frameworks like TensorFlow Graph data-dependent control flow must be embedded in the dataflow graph that will be executed.

Back-End: TensorFlow. We convert the PyTorch RNN implementation above to TensorFlow Eager mode and TensorFlow Graph mode. Both of these translations require a Snek overload for function calls that rewrites those targeting the PyTorch API to the corresponding TensorFlow API calls. Consider the following implementation of `max_`, which replaces a call to the PyTorch `torch.max` function with a call to TensorFlow's `tf.reduce_max` instead:

[8]While the evaluation was performed using Python 2.7.1, Snek uses libraries which allow either Python 2 or Python 3 code in a way that is transparent to users of Snek.

```
call = RewritingCallOverload(default_call)
@call.replaces(torch.max)
def max_(input_data):
  return tf.reduce_max(input_data)
```

The `RewritingCallOverload.replaces` decorator, a helper API provided by Snek, does the heavy lifting. This decorator registers the `max_` function as a replacement for `torch.max`. The argument to `RewritingCallOverload` tells the virtualization code what to do when a replacement function has not been registered, usually just call it as normal.

While this example is straightforward, a direct replacement for a given function will not always exist. In such cases, the domain expert implementing the virtualized functions is able to provide a conversion. For example the PyTorch and TensorFlow have different APIs for computing a permutation of a tensor. Our implementation of the conversion function for `torch.transpose` is below:

```
@call.replaces(torch.transpose)
def transpose(x, dim0, dim1):
  perm = []
  for i in range(x.shape.rank):
    if i == dim0:
      perm.append(dim1)
    elif i == dim1:
      perm.append(dim0)
    else:
      perm.append(i)
  return tf.transpose(x, perm)
```

Once we have defined the translation of API calls we can translate from PyTorch to TensorFlow Eager mode. Both of these frameworks execute control flow within the Python interpreter calling into their respective APIs as needed to compute any values needed to make control flow decisions. TensorFlow Graph mode works differently, first building up a dataflow graph representing a computation. Then giving the whole graph over to an executor which then computes the results. This means that any data-dependent control flow must be embedded in this dataflow graph. TensorFlow does not have a for loop so we override `for_stmt` with code that produces the appropriate `tf.while_loop` operation when the iterated object is a TensorFlow tensor.

Back-End: NumPy. The translation to NumPy corresponds directly with the translation to TensorFlow Eager. We use the same call rewriting mechanism to replace PyTorch API calls with NumPy API calls.

6.1.2 TensorFlow Eager. Our TensorFlow Eager implementation is similar to the PyTorch implementation seen above with calls to the TensorFlow API in place of the calls to the PyTorch API. We then translate this implementation to PyTorch, NumPy, and TensorFlow Graph. We only discuss this last translation as the others directly follow from the description in Section 6.1.1.

Comparison with AutoGraph. TensorFlow Eager to TensorFlow Graph is the same translation performed by TensorFlow AutoGraph. Our goal is not to reproduce a production-level system like AutoGraph in Snek, but to show that we can replicate its key features quickly and easily. While AutoGraph converts many features of Python, we compare to the recommended configuration or AutoGraph that includes conversions of `if`, `for`, and `while` statements.[9] Our Snek translation also fully supports the conversion of these three key control flow constructs. Below is an override of `if_stmt` as implemented in Snek. We use the helper function `execute_isolated` as described in Section 5.2.

```
def if_stmt(cond, body, orelse, local_writes):
  cond_result = cond()
  if tf.is_tensor(cond_result):
    def if_body(*_):
```

[9] https://www.tensorflow.org/alpha/guide/autograph

```
      modified_vals, _ = execute_isolated(body, local_writes)
      return modified_vals

    def if_orelse(*_):
      modified_vals, _ = execute_isolated(orelse, local_writes)
      return modified_vals

    result_values = _tf_if_stmt(cond_result, if_body, if_orelse)

    for var, val in zip(local_writes, result_values):
      var.val = val
  else:
    py_defaults.if_stmt(lambda: cond_result, body, orelse, local_writes)
```

Running the AutoGraph in the recommended configuration on our simple RNN example produced a throughput of 106.22 iterations per second which is comparable to the 105.70 iterations per second we get with Snek.

6.1.3 NumPy. We implemented the same RNN in NumPy with translations to PyTorch, TensorFlow Eager, and TensorFlow Graph. These follow the same pattern as those described in Section 6.1.1.

6.2 MNIST: PyTorch to Lantern

```
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(784, 50)
    self.fc2 = nn.Linear(50, 10)
    self.activateFunc = args.activateFunc

def forward(self, x):
  x1 = x.view([-1, 784])

  if self.activateFunc == 1:
    x2 = F.relu(self.fc1(x1))
    x3 = self.fc2(x2)
    x4 = F.log_softmax(x3, dim=1)
    return x4
  else:
    x6 = F.tanh(self.fc1(x1))
    x7 = self.fc2(x6)
    x8 = F.log_softmax(x7, dim=1)
    return x8
```

Fig. 12. Simplified implementation of MNIST in PyTorch, with optional hyperparameter to specify activation function.

Using Snek, we built a system capable of taking PyTorch code as input, and generating s-expressions representing this program to produce back-end agnostic code. We then used the parser provided by Amin et al. [2015] to convert these s-expressions into a Scala AST upon which Lantern may reason. Due to the fact that Lantern is built using Lightweight Modular Staging (LMS) [Rompf 2012], this provides additional opportunities for staging benefits beyond those provided by Snek.

We note that Lantern does not have a Python front-end; rather, we are able to easily construct one using Snek to generate code, rather than dispatching to a staged computation. Furthermore, by generating s-expressions, we can effectively create a Python front-end for *any* back-end system, though this requires the writing of a parser capable of translating s-expressions to the desired system. This requirement, however, becomes the responsibility of the back-end developer: the DSL developer must only generate correct s-expressions, and the end user must only construct a valid program. Indeed, with this pattern the back-end developer need not develop in Python, and may instead implement a parser in the language of their choice.

Evaluating Snek and Lantern. In order to evaluate Snek and Lantern, we elected to convert the code in Figure 12. This is a simplified (for presentation) implementation adapted from the

implementation provided by PyTorch[10], using a single fully-connected layer (consisting of two `Linear` layers). This is a program which performs image classification on the MNIST [LeCun and Cortes 2010] dataset, which is a standard introductory program for machine learning programmers. MNIST consists of 70,000 handwritten digits (60,000 training examples, 10,000 test examples), which machine learning models aim to learn to classify correctly (i.e., correctly identifying the number pictured in an image). We elide all training details here for presentation.

Results. In order to provide a baseline for Snek, we implemented the equivalent program directly in Lantern. In both cases, Lantern generates a C++ file containing the staged functionality. Running these for both systems yields *identical* results, with both achieving an average of 21.5 seconds to train 5 epochs (average of 5 runs reported). The accuracy was similarly unaffected. Running this code in unmodified PyTorch code takes 122.4 seconds, while using Snek targeting PyTorch takes 124 seconds.

Due to the fact that Lantern heavily specializes the generated code, this result is unsurprising: a faithful translation from Python will yield the same C++ code as that from Scala, as there do not exist additional specialization opportunities.

7 RELATED WORK

DSLs in Python. Despite the abundance of DSLs which exist in Python, very few frameworks exist which aim to aid in the creation of new Python DSLs. One such framework is MacroPy [MacroPy3 Contributors 2019], which allows for user-defined macros to modify Python ASTs at import time. This allows for easy extension of the default Python semantics, but requires a modified interpreter, and does not provide any staging functionality due to its static behavior. Marrow [Marrow Contributors 2019b] similarly performs all translation at import time. Other systems like cinje [Marrow Contributors 2019a], Django [Django Contributors 2019], and Jinja [Jinja Contributors 2019] perform Python templating which allows for embedding DSLs in Python, but does not allow for the virtualization of Python constructs to provide new semantics. In particular, these systems require a new parser to be used, rather than that of the Python interpreter.

Language-oriented Programming. Language-oriented programming is a programming paradigm in which domain experts may design a DSL to solve a problem, rather than using a general-purpose one. Racket [Felleisen et al. 2015] achieves this through an advanced macro system, and provides users with an easy mechanism for designing new language constructs (both syntactic and semantic). While the present work was designed following the tenets of language-oriented programming as outlined by Felleisen et al. [2018], the proposed methodology intentionally does not provide a mechanism for introducing new syntax. In short: while both Racket and Snek provide users with the powerful abstractions available via DSLs [Hudak 1997], Racket is used for the construction of new DSLs, whereas Snek repurposes Python *as* a DSL.

Multi-stage Programming. Multi-stage programming [Taha and Sheard 2000] is a well-studied area. Tools like Terra [DeVito et al. 2013] showcase the ability to metaprogram to an intermediate language specifically designed to integrate between user-facing code and highly-optimized machine code. Similarly, code quotations in F# [Cisternino et al. 2007] allow for the explicit construction of ASTs of the host language for later execution, including the ability to reference existing code objects.

Of most relevance to Snek is Lightweight Modular Staging (LMS) [Rompf and Odersky 2012], upon which the system presented in Section 6.2 is based. LMS uses a specialized type annotation `Rep[T]` to allow users to mark values as being known at run time, with all other values known at

[10]https://github.com/pytorch/examples/blob/master/mnist/main.py

compile time, and relies on virtualization of Scala built-ins [Rompf et al. 2012]. A number of existing works have shown LMS's ability to provide users with an extremely high-level interface while generating highly specialized low-level code [Essertel et al. 2018; Rompf et al. 2013, 2014; Stojanov et al. 2018; Tahboub et al. 2018]. Of most relevance here is Lantern [Wang et al. 2018; Wang and Rompf 2018], which uses LMS to provide a differentiable programming interface. Amin and Rompf [2018] also showed how multi-stage programming can be used to reduce the overhead inherent in different interpreter boundaries.

These systems all rely on static type information, whereas Snek is the first system to provide such capabilities in a dynamically typed setting.

Macros and other Dynamic Languages. Virtualization in the style of Snek is crucially different from macros in the style of Lisp, the prevalent metaprogramming technique in dynamic languages: a macro receives unevaluated pieces of a host-language AST as arguments. A staging system based on virtualized functions receives staged code fragments that are *the result of* host-language evaluation. For DSL developers, this makes the difference between having to analyze host-language ASTs to extract DSL expressions (a difficult task!) and receiving proper DSL expressions as arguments right away.

However, syntactic macros with AST introspection and rewriting facilities appear to be enough to implement Snek-style virtualization in other languages — in essence, the approach boils down to rewriting built-in primitives into function calls that constitute a tagless-final [Carette et al. 2007] or object algebra [d. S. Oliveira and Cook 2012; Guttag and Horning 1978] representation of the language syntax. Taking the role of decorators in Python, macros could be used to implement Snek's source-to-source transform in Racket or other Lisp dialects, which would then enable a similar form of virtualization in those languages. Likewise, the same approach could be applied to Java using Recaf [Biboudis et al. 2016], or to JavaScript using a macro toolkit such as SweetJS [Sweet.js Contributors 2019].

Partial Evaluation. Partial evaluation is closely related to multi-stage programming: both are specialization approaches, but partial evaluation aims to be entirely automatic in this specialization [Jones et al. 1993]. Snek attempts to specialize based on user intent, with this intent expressed through the use of decorators.

Python Semantics. λ_π [Politz et al. 2013] is a formal system describing the core semantics in Python as an executable small-step operational semantics. The work of λ_π presents a number of features within Python worthy of examination, especially in works like Snek which perform source code transformations. λ_π is not a formal proof of Python semantics, nor is conformance to λ_π a guarantee of a correct model of Python semantics. However, λ_π exposes an extensive test suite modeled after the Python `unittest` suite, including a number of tests which examine many non-evident features in Python (e.g., functions declared within `class` definitions *not* having access to variables declared within the enclosing `class` without the use of `self`). Other works include an executable operational semantics for a subset of Python in Haskell [Guth 2013], as well as in the K semantic framework [Guth 2013; Rosu and Serbanuta 2010].

8 CONCLUSIONS

We presented a general-purpose methodology for customizing the semantics of syntactic constructs to enable linguistic reuse of Python. It enables deeply embedded DSLs in Python, and "staging based on types," without requiring additional type information. This is accomplished through virtualization of Python's built-in constructs which could not be otherwise overloaded, thus providing a new stage for computation at which time types are known. We demonstrated these capabilities in a

system called Snek, by providing sample implementations of typical DSLs. Virtualization as shown in Snek unlocks the full expressive power of Python in DSLs, and can target arbitrary back-ends, whether in Python or otherwise.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR* abs/1605.02688 (2016).

Nada Amin et al. 2015. LMS Black aka Purple. https://github.com/namin/lms-black.

Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *PACMPL* 2, POPL (2018), 52:1–52:33.

Andrew W. Appel and Trevor Jim. 1989. Continuation-Passing, Closure-Passing Style. In *POPL*. ACM Press, 293–302.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science Engineering* 13, 2 (2011), 31 –39. https://doi.org/10.1109/MCSE.2010.118

Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. 2016. Recaf: Java dialects as libraries. In *GPCE*. ACM, 2–13.

R. Bradshaw, S.Behnel, D. S. Seljebotn, G.Ewing, and et al. 2011. The Cython compiler. http://cython.org. Accessed: 2018-11-05.

Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. *20th International Conference on Parallel Architectures and Compilation Techniques*.

Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *APLAS*. 222–238.

H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. 2010. Language Virtualization for Heterogeneous Parallel Computing *(Onward!)*.

H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. 2011. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*.

Antonio Cisternino, Adam Granicz, and Don Syme. 2007. *Expert F#*. Apress.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP (Lecture Notes in Computer Science)*, Vol. 7313. Springer, 2–27.

Dask Contributors. 2019. Dask. https://github.com/dask/dask. Accessed: 2019-04-05.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: a multi-stage language for high-performance computing. In *PLDI*. ACM, 105–116.

Django Contributors. 2019. Django. https://www.djangoproject.com/. Accessed: 2019-04-05.

Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. USENIX Association, 799–815.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *SNAPL (LIPIcs)*, Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 113–128.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (2018), 62–71.

Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax

Dwight Guth. 2013. *A Formal Semantics of Python 3.3.* Master's thesis. University of Illinois at Urbana-Champaign.

John V. Guttag and James J. Horning. 1978. The Algebraic Specification of Abstract Data Types. *Acta Inf.* 10 (1978), 27–52.

C. Hofer, K. Ostermann, T. Rendel, and A. Moors. 2008. Polymorphic embedding of DSLs *(GPCE)*.

Paul Hudak. 1997. Handbook of Programming Languages, Volume III Little Languages and Tools. (1997).

Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dongjin Shin, and Byung-Gon Chun. 2019. JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs. In *NSDI*. USENIX Association, 453–468.

Jinja Contributors. 2019. jinja. http://jinja.pocoo.org/. Accessed: 2019-04-05.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*. ACM, 1–12.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

MacroPy3 Contributors. 2019. MacroPy3. https://github.com/lihaoyi/macropy. Accessed: 2019-04-05.

Marrow Contributors. 2019a. cinje. https://github.com/marrow/cinje. Accessed: 2019-04-05.

Marrow Contributors. 2019b. Marrow DSL. https://github.com/marrow/dsl. Accessed: 2019-04-05.

Microsoft Research. 2019. Z3. https://github.com/Z3Prover/z3. Accessed: 2019-04-05.

Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2018. AutoGraph: Imperative-style Coding with Graph-based Performance. *CoRR* abs/1810.08061 (2018).

Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *SysML*.

Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation (PEPM)*.

Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018), 1002–1015.

Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 217–232. https://doi.org/10.1145/2509136.2509536

PyTorch Contributors. 2018. Torch Script. https://pytorch.org/docs/master/jit.html. Accessed: 2018-09-24.

PyTorch Contributors. 2019. PyTorch. https://pytorch.org/. Accessed: 2019-04-05.

Tiark Rompf. 2012. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming.* Ph.D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-5456

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* 25, 1 (2012), 165–207.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Conference on Generative programming and component engineering (GPCE)*. 127–136. https://doi.org/10.1145/1868294.1868314

Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-Level Programs *(POPL)*.

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *PLDI*. ACM, 41–52.

Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebr. Program.* 79, 6 (2010), 397–434.

RustPython Contributors. 2019. RustPython. https://github.com/RustPython/RustPython. Accessed: 2019-04-05.

Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD intrinsics on managed language runtimes. In *CGO*. ACM, 2–15.

Sweet.js Contributors. 2019. Sweet.js. https://www.sweetjs.org/. Accessed: 2019-07-05.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engineering* 13, 2 (2011), 22–30.

Bart van Merrienboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. In *Advances in neural information processing systems*.

Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. 2018. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *NIPS*.

Fei Wang and Tiark Rompf. 2018. A Language and Compiler View on Differentiable Programming. *ICLR Workshop Track* (2018). https://openreview.net/forum?id=SJxJtYkPG