The 800 Pound Python in the Machine Learning Room

James M. Decker[1], Dan Moldovan[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)

Abstract

Modern machine learning frameworks have one commonality: the primary interface, for better or worse, is Python. Python is widely appreciated for its low barrier of entry due to its high-level built-ins and use of dynamic typing. However, these same features are also often attributed to causing the significant performance gap between the front-end in which users are asked to develop, and the highly-optimized back-end kernels which are ultimately called (generally written in a lower-level language like C). This has led to frameworks like TensorFlow requiring programs which consist almost entirely of API calls, with the appearance of only coincidentally being implemented in Python, the language.

All recent ML frameworks have recognized this gap between usability and performance as a problem and aim to bridge the gap in generally one of two ways. In the case of tools like PyTorch's JIT compiler, executed tensor operations can be recorded via tracing based on operator overloading. In the case of tools like PyTorch's Torch Script, Python functions can be marked for translation entirely to a low-level language. However, both tracing and wholesale translation in this fashion have significant downsides in the respective inability to capture data-dependent control flow and the missed opportunities for optimization via execution while a low-level IR is built up.

In this paper, we demonstrate the ability to overcome these shortcomings by performing a relatively simple source-to-source transformation, that allows for operator overloading techniques to be extended to language built-ins, including control flow operators, function definitions, etc.

We utilize a preexisting PLT Redex implementation of Python's core grammar in order to provide assurances that our transformations are semantics preserving with regard to standard Python. We then instantiate our overloading approach to generate code, which enables a form of multi-stage programming in Python. We capture the required transformations in a proof-of-concept, back-end agnostic, system dubbed Snek, and demonstrate their use in a production system released as part of TensorFlow, called AutoGraph. Finally, we provide an empirical evaluation of these systems and show performance benefits even with existing systems like TensorFlow, Torch Script, and Lantern as back-ends.

1 Introduction

Python remains the language of choice for machine learning practitioners. Due to Python's high-level interface and "beginner friendly" dynamic typing system which provide a relatively low barrier of entry, the performance detriments are largely seen as a necessary trade-off for wider accessibility. Even proposals like Swift for TensorFlow [32], which bridge this gap as well as providing a number of other static analysis benefits, have not yet been widely adopted due to the effort and expense required in migrating to a new language or framework.

Many machine learning frameworks targeting Python were initially designed under the perception that there exists a strict and unavoidable dichotomy that such a system must be either easy to use, *xor* performant. PyTorch [20], for example, was developed with the goals of interactivity and ease-of-expression first, thus foregoing opportunities for whole-program optimization. On the other side of this perceived fence are systems like TensorFlow [1]. TensorFlow programs consist almost entirely of API calls (in an effort to involve the Python interpreter as little as possible) which build a computation graph for later execution.

This dichotomy is untenable for users, and is one which we aim to resolve. Indeed, PyTorch, TensorFlow, and others are now exploring mechanisms by which users may write code in idiomatic Python, without the expected performance loss incurred from the Python interpreter [2]. These efforts tend towards one of two solutions. The first is to translate entire Python ASTs to another language; the second is tracing via operator overloading. However, neither solution is without its flaws, ultimately leading to missed optimization opportunities or usability concerns.

Looking beyond Python, we can see that many of the problems posed have already been solved in statically-typed languages. Of particular relevance is Lightweight Modular Staging (LMS) [24], which provides users the ability to do multi-stage programming in Scala. LMS uses "staging based on types," exposing a type annotation to users to explicitly mark computations for current or future execution: `Rep[T]` types will generate code; all other types will be executed as normal. This is similar to tracing with the added ability to capture data-dependent control flow, as well as providing native code generation [10]. The capabilities provided by LMS meet all of the requirements of a machine learning audience except one: it is unavailable in Python [39].

Existing efforts such as Torch Script [22] aim to provide a high-level interface for users, while ultimately generating a computation graph of user programs. Efforts mix tracing methods with a translation of idiomatic Python to a Python subset (Torch Script), ultimately generating code. Such efforts generally rely on Python's mechanism for metaprogamming: decorators. Decorators in Python are function annotations which allow for arbitrary code to be evaluated both at

Preprint, November 2018.

the time of function definition and at each function invocation.

However, current efforts are not always informed by proper Python semantics, thus having no guarantees of correctness beyond the developers' intuition. Furthermore, these efforts in many cases miss optimization opportunities due to a lack of generality. A key example of this can be seen in efforts which perform tracing (e.g., Torch Script, Open Neural Network eXchange (ONNX) [18]): such methods lose all information regarding control flow in the generated code.

In this paper, we examine the metaprogramming capabilities provided by Python, and utilize decorators to enable multi-stage programming in Python. The key insight is that a decorator inherently breaks decorated functions into two stages: one at function definition, another at function invocation. This allows for manipulation of the function body code upon definition, and allowing for a specialized execution at function invocation (including code generation).

We provide a set of source code transformations able to enable generative programming in Python. We implement these transformations in a system called Snek, and use a core grammar of Python (λ_π [21]) to provide assurances of semantic preservation. We then extend these transformations to enable multi-stage programming in the style of Lightweight Modular Staging, targeting (and implemented entirely in) Python. We further describe the challenges of implementing a system "based on types" in a dynamically-typed language, as well as other challenges which arise from differences between Scala and Python (i.e., Python's use of statements vs. Scala's restriction of expressions, some Python-specific scoping rules, etc.). This notably does not require any additional compiler plug-ins or modifications to the Python interpreter. Snek is also back-end agnostic, ultimately generating S-Expressions capable of being easily parsed by any system. To illustrate this, we target both Torch Script and the Lantern engine [38] as back-ends, using a Lisp parser written in Scala for interfacing with Lantern. We also show the use of these transformations in a production system, AutoGraph, in which the generation of S-Expression is bypassed in favor of directly generating TensorFlow API calls. AutoGraph also utilizes more sophisticated analysis methods in order to better inform more specialized code generation; we discuss these in Section 5.7. We note that AutoGraph is slated to be incorporated in the TensorFlow 2.0 release.[1]

This paper makes the following contributions:

- We examine the techniques currently in use to bridge the ease-of-use and performance gap in Python, and show the need for source code transformations in addition to these techniques (Section 2).
- We present a series of source code transformations targeting Python, including providing Scala-style virtualization for control flow operators (Section 3), and introduce Snek, an implementation of these transformations.
- We adopt the formal semantics defined in λ_π, a grammar which captures the core semantics of Python [21], and formally present our virtualization function, $[\![\ ]\!]_v$, in reduction semantics. We further present a semantic preservation property as a convergence relation to λ_π (Section 4).
- We extend our transformations to provide staging in Python in the manner of Lightweight Modular Staging, utilizing Python's dynamic dispatch to resolve type information, and introduce Snek: an implementation of these transformations and staging capabilities (Section 5). We also discuss a production system, AutoGraph, built using transformations in the style of Snek, and show the design decisions which may change when targeting a single back-end.
- We evaluate Snek and AutoGraph, comparing with current state-of-the-art systems (Section 7).

[1]https://pgaleone.eu/tensorflow/gan/2018/11/04/tensorflow-2-models-migration-and-new-design/

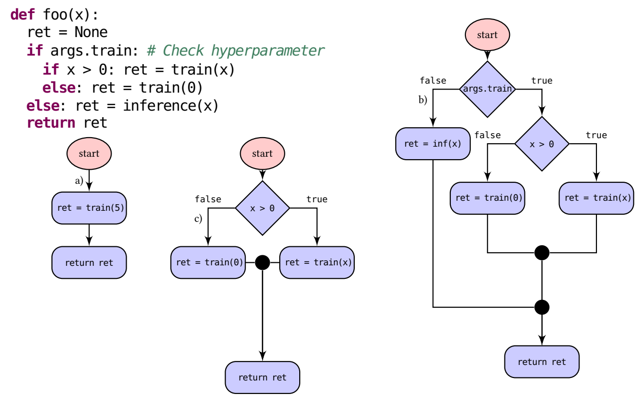

Figure 1. Control Flow Graphs generated from function `foo` (top) using a) `@torch.jit.trace` with a sample value of `x = 5` and `args.train` set to `True`, b) Snek with `args.train` set to `True`, and c) `@torch.jit.script`.

2 Translation, Tracing, and the Need for Virtualization

We begin by examining existing techniques used to allow users to program in idiomatic Python, while still achieving competitive performance. We focus primarily on systems which perform either source-to-source (STS) translation or operation tracing, paying special attention to tools like Torch Script which provide these techniques in the setting of machine learning applications.

2.1 AST Translation

A number of existing systems [5, 14, 19, 22, 35] make use of STS translation in order to bypass the overhead inherent in the Python interpreter [2] altogether. These instead operate on Python ASTs, which are then translated to a different, generally lower-level, language. We examine two systems which perform STS translation targeting Python here, and provide a more exhaustive analysis in Section 8.

Cython. Perhaps the most well-known Python translation tool is Cython [5]. 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[2] in Figure 2 (left) as input, Cython will produce a .c file which runs approximately 35% faster than in Python [6]. 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 by supplying type annotations, as show in Figure 2.

```
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)

Figure 2. Cython tutorial code with (left) and without (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 [6].

While these results are impressive, even in the trivial example shown, there may 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.

***Torch Script:* `torch.jit.script`.** Translations of the nature depicted here are desirable when looking to capture data-dependent control flow in the original user code. PyTorch's Torch Script [22] framework provides a translation mechanism in the form of a new decorator: `@torch.jit.script` (we refer to this as `@script` for the remainder of the paper). Torch Script's `@script` decorator behaves similarly to Numba [14]: it takes as input a Python function which will be interpreted as a specialized subset of Python, in this case, Torch Script. Whereas tools like Cython typically require translation of entire programs (and, in the case of any errors, may require users to be proficient in the generated language), `@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). Torch Script is intended to be used in accomplishing machine learning tasks, and provides the benefit of allowing users to more easily save models for later use in the form of a computation graph.

In order to achieve this flexibility, Torch Script (at the time of writing) imposes a number of limitations upon users. Of particular relevance here are the limitations that all functions decorated with `@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 `a_num` is a `Number`, rather than a `tensor` value:

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

Furthermore, although Torch Script in its current iteration does provide the benefit of increased usability for users, as with Cython, `@script`'s current method of translation does not utilize any data *independent* information. Consider, for example, the function in Figure 1 (top). This produces a computation graph expressible as the control flow graph in Figure 1 (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 Torch Script already implements some dead code elimination through the use of liveness analysis, but opportunities such as this are currently overlooked.

Multi-stage programming. Cython, Torch Script, and similar systems which perform wholesale translation in this fashion fail 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 [23, 24] *in addition* to the translation techniques currently used by systems like Cython and Torch Script.

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 [20].

***Torch Script:* `torch.jit.trace`.** PyTorch [20] performs tracing in this manner, 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 Torch Script framework. Like `@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 trace. Tracing in this fashion is typically accomplished via operator overloading,

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

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. Consider the code in Figure 1 (top). Invoking `trace` on `foo` yields the control flow graph shown in Figure 1 (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.

Language Virtualization. Simply overloading operators is not enough to capture all relevant information: overloading of control flow constructs is also required. Chafi et al. [7] proposed to "virtualize" such built-in language features 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. Such virtualization is not immediately available in Python: there is no notion of overloading a magic `__if__` method as with many constructs. Instead, we propose extending the operator overloading found in systems like PyTorch to also include such virtualization through the use of source code transformations on the original program, effectively choosing to generate Python (rather than e.g., Torch Script) via preprocessing. In this generated, intermediate Python, we aim to produce constructs which will ultimately generate a computation graph through the use of operator overloading, based on the type(s) of the operand(s). Performing virtualization in this manner allows for data *independent* control flow to be removed in any extracted computation graphs, while still capturing data *dependent* control flow, as shown in Figure 1 (b).

This technique is similar to the notion of "staging based on types" exhibited by systems such as Lightweight Modular Staging [25]. Such type-based staging efforts rely heavily on having a static type system; in a dynamically typed language like Python, however, it becomes impossible to know statically which operations will generate code. Consider, for example, the following Python function:

```
def bar(n):
  x = 0
  while x < n: x = x + 1
  return x
```

Here, if `n` can be known during the tracing phase of our staging (also called "current" stage), the value of `x` will also be known: it may be an unmodified Python integer. However, if `n`'s value will only become known at a future stage, we must have some custom type to not only represent `n`, but we must also assign `x` this type. Failing to do so would cause `x = x + 1` to evaluate a single time, which would lead to an incorrect result in nearly all cases.

3 Snek: Python Generating Python

Having determined the necessity for virtualization (with a desire to ultimately enable staging in Python), we now examine how such virtualizations may be built. We create a virtualization function $[\![s]\!]_v$ which takes some Python statement *s* and virtualizes it according to the rules in Figure 3, taking care to preserve the original semantics (see Section 4). We devote the remainder of this section to the further explanation of these virtualization rules, paying special attention to examine those features of Python which require additional consideration.

$[\![$ `if cond:` s_1 `else:` s_2 $]\!]_v$	=	`def then$1():` $[\![s_1]\!]_v$ `def else$1():` $[\![s_2]\!]_v$ `_if(`$[\![\text{cond}]\!]_v$`,then$1,else$1)`
$[\![$ `while cond:` s_1 $]\!]_v$	=	`def body$1():` $[\![s_1]\!]_v$ `def cond$1(): return` $[\![\text{cond}]\!]_v$ `_while(cond$1, body$1)`
$[\![$ `for i in` s_1`:` s_2 $]\!]_v$	=	`def body$1(i):` $[\![s_2]\!]_v$ `_for(`$[\![e_1]\!]_v$`, body$1)`

$[\![$
`def fun(`$p_1,\ldots,p_n$`):` s_1
$]\!]_v$
	=	`def fun(`$p_1,\ldots,p_n$`):`
		  `try:` $[\![s_1]\!]_v$
		  `except NonLocalReturnValue as r:`
		    `return r.value`

$[\![$`return` $s_1]\!]_v$	=	`_return(`$[\![s_1]\!]_v$`)`
$[\![$ `x =` $e_1]\!]_v$	=	`x = _var()` `_assign(x,` $[\![e_1]\!]_v$`)`
$[\![$ `x` $]\!]_v$	=	`_read(x)`
$[\![\_]\!]_v$	=	`_`

Figure 3. Virtualization rules in Snek. Note that we use fresh names for all extracted function names. Rules modifying variables apply only to those which require lifting (i.e., multiple assignment). All statements listed may represent multiple statements; we elide proper Python whitespace rules for presentation only.

3.1 Virtualizing `if/else`

In virtualizing `if/else` statements, we have two statements we need to transform. We elect to extract the `then` and `else` branches (`e$_1$` and `e$_2$`, respectively) into standalone functions, though the conditional `cond` does not require any transformation. The entire `if` block is then replaced with the extracted functions, followed by a call to our virtualized `_if` function, shown here:

```
def _if(test, body, orelse):
  if test: return body()
  else: return orelse()
```

However, consider the following program:

```
x = my_int_fun() # returns some integer
cond = my_fun() # returns some bool
if cond: x = x + 1
else: x = x - 1
```

Transforming this program via function extraction in the manner described yields the following:

```
x = my_int_fun() # returns some integer
cond = my_fun() # returns some bool
def then$1(): x = x - 1
def else$1(): x = x + 1
_if(cond, then$1, else$1)
```

This code is semantically incorrect, and causes the Python interpreter to exit with an exception that `x` has been used without having been initialized (in either `then$1` or `else$1`, depending on the value of `x`). This is due to the fact that in Python, functions have implicit permission to read all variables accessible in the containing scope, but do *not* have permission to write to them. As such, Python views the bodies of the extracted functions as attempting to define a new variable x, rather than updating the x provided as a parameter of foo.

To resolve this, we choose to lift variables which behave as mutable variables (i.e., do not satisfy SSA). Snek contains a context which tracks lifted variables in order to comply with proper Python semantics (and generated the appropriate code). Note that we determine the necessary variables *before* performing any transformations. With lifting in place, our transformed foo is as follows:

```
x0 = my_int_fun()
x = _var()
_assign(x, x0)
cond = my_fun()
def then$1(): _assign(x, _read(x) - 1)
def else$1(): _assign(x, _read(x) + 1)
_if(cond, then$1, else$1)
```

We note that this is not specific to virtualization of if/else statements; we encounter such a problem with all translations which extract functions. We thus apply lifting in each of these cases, though we elide these details when discussing future transformations.

3.2 Virtualizing while

Similar to if, virtualizing a while loop requires the extraction of a function containing the body of the loop (body$1). However, due to the fact that the condition (cond) may be evaluated multiple times (potentially with a different result), we must also extract a function returning the result of evaluating the conditional (cond$1). The original while construct is then replaced with the extracted functions, followed by a call to the virtualized _while function, shown below:

```
def _while(test, body):
  ret = None
  while test() and ret is None: ret = body()
  return ret
```

3.3 Virtualizing for

The virtualization of for follows mechanically, with the only notable difference being the requirement that we must add a parameter to the extracted function representing the loop body. Note that this parameter must have the same name as the original iteration variable: we accomplish this by extracting the name at transformation time. We present _for, as follows: `def _for(it, body): for i in it: body(i)`

Generalizing this beyond a single iteration variable (i.e., implicit object deconstruction) is trivial; we elide these details for a cleaner presentation.

3.4 Virtualizing Function Definitions

We encounter some difficulty in virtualizing function definitions due to our need to propagate return values from generated functions. As an example, consider the following function: `def foo(x): if x > 0: return 1 else: return 0`

Transforming the body of foo using only the if/else rules in Figure 3 results in the following:

```
def foo(x):
  def then$1(): return 1
  def else$1(): return 0
  _if((x > 0), then$1, else$1)
```

While the extracted functions contain return statements, these values will not be returned from foo. Upon first glance, it seems the solution is to wrap all calls to _if in a return.

```
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')
```

Figure 4. Example of nested function scopes in Python (left) and the effect of nonlocal (right). Originally appeared in Politz et al. [21].

However, the astute reader will note that while this is sufficient in the trivial example shown, in the general case this would lead to functions returning prematurely, as if statements need not contain return statements. Thus, it becomes necessary to introduce the notion of a *nonlocal return value*, which may arise at any point in execution and be handled by the containing scope. Python contains a construct with the desired semantics: Exceptions. We introduce the following class:

```
class NonLocalReturnValue(Exception):
  def __init__(self, value): self.value = value
```

We then virtualize all return statements, with _return defined as follows: `def _return(value): raise NonLocalReturnValue(value)`

Finally, in order to "catch" the return value, we surround the function body with a try/except block. Correctly transformed, then, our trivial example is as follows:

```
def foo(x):
  try:
    def then$1(): _return(1)
    def else$1(): _return(0)
    _if(x > 0, then$1, else$1)
  except NonLocalReturnValue as r: return r.value
```

3.5 Introducing @lms

To provide these transformations to users with minimal modifications, Snek provides a decorator, @lms, which serves as the entry point for Snek's metaprogramming capabilities. @lms uses a custom ast.NodeTransformer object to perform in-place modifications on the ASTs extracted from user code. As shown in Section 2, use of a decorator is consistent with the current state-of-the-art production systems due to their ease of use and ability to be used at the granularity of functions. Snek may be configured such that the entirety of a user's program is wrapped within a dummy function and transformed, though this becomes undesirable with the addition of staging (Section 5).

4 Preservation of Semantics

We wish to have some assurance that these virtualizations are semantics- preserving for all programs without staged values. In this section, we present elements of the Python semantics which pose some difficulty in transforming in the manner hitherto presented, and show a formal correspondence using reduction semantics that all transformations in $[\![\ ]\!]_v$ have this desired semantic preservation property.

4.1 Scope

In perhaps the most comprehensive formal PL view of Python to date, Politz et al. [21] demonstrate a number of features in Python's semantics which may appear unintuitive to many users. Python contains three types of variables in relation to scoping rules: `global`, `nonlocal`, and `local`, with the majority of identifiers falling into the `local` category. A simplified view is simply that all scopes have read-only access to all variables declared in any enclosing scopes. 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).

Snek does not currently allow for variable shadowing (and, therefore, `nonlocal` and `global` declarations), but this is planned for a future release.

4.2 The Full Monty

λ_π as presented by Politz et al. [21] is an executable small-step operational semantics written in PLT Redex [11] 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 4.1, 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` a `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 $\rightarrow^*$. Snek does not currently allow for variable shadowing: we thus assume that all variables in the Python expression e must have fresh names.

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:

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

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).

```
(→  ((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")
```

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

5 Multi-Stage Programming

With the virtualizations described in Section 3 in place, we now have the ability to overload the functionality of built-in operations in Python. However, these transformations alone do not provide any notable benefit to users. As we have modeled our virtualization rules after those found in Lightweight Modular Staging [17, 24], we may now turn our attention to incorporating the multi-stage programming capabilities offered there.

5.1 Lightweight Modular Staging

Lightweight Modular Staging (LMS) [23, 24] is a multi- stage programming framework which enables "staging based on types." LMS provides users with a type annotation `Rep` which

[3] Python version 3.2.3

[4] https://github.com/brownplt/lambda-py

```
@lms
def run(x):
  def power(n, k):
    if k == 0: return 1
    else:
      return n *
        power(n, k - 1)
  return power(x, 2)
```

```
def run(x):
  try:
    def power(n, k):
      try:
        def then$1():
          _return(1)
        def else$1():
          _return((n * power(n, (k - 1))))
        _if((k == 0), then$1, else$1)
      except NonLocalReturnValue as r: return r.value
    _return(power(x, 2))
  except NonLocalReturnValue as r: return r.value
```

```
(def runX (in1)
  (begin
    (let x0 (* in1 1)
    (let x1 (* in1 x0)
      x1))))
```

Figure 6. Python implementation of power with base staged and exponent fixed to 2 (left), generated Python IR (middle), and resultant S-Expr (top-right).

allows the explicit demarcation of objects which will generate code (i.e., may not be known at compile time). LMS uses advanced operator overloading to accomplish this code generation: types marked with the Rep annotation generate (highly specialized) code at each operation, with values known at the time of compilation becoming constant values in the generated code. Notably, the result of any computation involving a Rep value must always be a Rep value, but any value known at compile time may be lifted to a Rep as needed.

5.2 Staging Based on Types...Without Types?

```
def addOne(x): return x + 1
a = 1; b = Rep('in'); c = 2.0
addOne(a) # → 2
addOne(b) # → in + 1
addOne(c) # → 3.0
```

```
(begin
  (let x0 (+ in 1)
  x0))
```

Figure 7. Generating code in pure idiomatic Python (left), and the resultant S-Expression (right).

While LMS has the functionality we wish to use, staging operations in LMS rely entirely on the use of static type information. This information is unavailable in a dynamic language, however.

One could add type annotations in Python: however, this is at odds with idiomatic Python as currently found in nearly all implementations of popular machine learning models. Indeed, this removes the dynamic typing capability which is core to Python. We require a solution which allows users to think of staging as one would expect in Python: *values* which are known at either compile- or runtime, rather than *types*. We introduce a new class Rep with the intent of overloading operations of this class to generate code in place of normal computation. In this manner, the original definition of addOne need not be modified: its behavior (as with every function in Python) is dependent on the value actually given to the function. In fact, this function can be used by any type which can perform addition with an integer, as shown in Figure 7 (left).

While this example is trivial, we provide an implementation of power in Figure 6 (left). Here, power is contained within a driver function run, which takes some parameter x. We perform the transformations described in Section 3, which results in the virtualized code shown in Figure 6 (center). Upon execution of this code, if x is of type Rep, Snek generates code for this type (shown in Figure 6, right).

5.3 Generating S-Expressions

In generating S-Expressions, we note that our transformed Python code satisfies SSA, with mutability expressed through the use of lifted variables (Section 3.1). Due to our initial design being heavily influenced by Scala, we elect to have all expressions in the generated S-Expression have a return value, with this value being the final value in a let-binding. To facilitate this, we define a function reflect capable of generating let-bindings (fresh returns a fresh variable name):

```
def reflect(s):
    global stBlock
    id = fresh()
    stBlock += [["let", id, s]]
    return id
```

We can thus define Rep as follows:

```
class Rep(object):
  def __init__(self, n): self.n = n
  def __add__(self, m): return reflect(["+",self,m])
  ... # other implementations are mechanical, we elide them here
```

With these in place, we are now able to generate code for simple programs, using the code in Figure 7 (center), ultimately generating the S-Expression in Figure 7 (right).

5.4 Staging Virtualized Functions

```
def _if(test, body, orelse):
  if not isinstance(test, Rep):
    if test: return body()
    else: return orelse()
  else:
    def capture(f):
      try: return (False, reify(f))
      except NonLocalReturnValue as e: return (True, e.value)
    thenret, thenp = capture(body)
    elseret, elsep = capture(orelse)
    rval = reflect(["if", test, thenp, elsep])
    if thenret & elseret: raise NonLocalReturnValue(rval)
    elif (not thenret) & (not elseret): return rval
    else:
      raise Exception('if/else: must return in all or no branches')
```

Figure 8. Virtualized function of Python if when adding staging functionality to generate S-Expression.

We present the modifications which must be made in our virtualized functions in order to enable staging in Figure 8. We note that the majority of these are mechanical, with the addition of a capture function. capture serves to propagate return statements through staged constructs, as well as to detect instances of return statements which are disallowed (i.e., in control flow structures).

5.5 Recursive Functions with Rep Conditionals

```
@rep_fun                          def f(a1,...,an):
def f(a1,...,an):        =          s1
  s1                              _def_staged(f, p1,...,pn)
f(p1,...,pn)                      _call_staged(f, p1,...,pn)
```

Figure 9. Transformation rules for staging functions.

Consider the implementation of power in Figure 6 (left). Due to the deferred execution of Rep values, calling this function with a Rep value for parameter k will yield an infinite chain of recursive calls, as then$\pyd$1 will never yield a value (see Figure 8, _if), and instead will generate code indefinitely. As such, all recursive functions which rely on a staged recursive conditional must also be staged. In order to provide such functionality, we introduce a new decorator, @rep_fun, which allows users to mark a function to be staged.

We present the staging transformations which must be added to support this in Figure 9, with the virtualized `_def_staged` and `_call_staged` as follows (`reflectDef` is a slightly modified `reflect` as shown in Section 5.3):

```
def _def_staged(f, *args):
    nargs = [fresh() for _ in args]
    return reflectDef(f.__name__, nargs, reify(f, *nargs))

def _call_staged(f, *args):
    return reflect([f.__name__, *args])
```

One interesting difficulty replacing function invocations with calls to `_def_staged` and `_call_staged` is that the context in which the invocation occurs may not allow for a simple transformation. Consider, for example, the following recursive implementation of `power`:

```
@rep_fun
def power(x, n):
  if n == 0: return 1
  else:
    return x * power(x, n - 1)
```

A naive replacement strategy yields the following generated Python code (we elide our other virtualizations for simplicity):

```
@rep_fun
def power(x, n):
  if n == 0: return 1
  else:
    return x * _def_staged(power,x,n-1)_call_staged(power,x,n-1)
```

As a current limitation of Snek, we simply require all calls to staged functions to be captured in a variable, as follows (note that Python does not perform tail recursion optimization[5]):

```
@rep_fun
def power(x, n):
  if n == 0: return 1
  else:
    ret = power(x, n - 1)
    return x * ret
```

We note for completeness that not all back-ends are capable of reasoning about recurrent functions: in tailoring a production system to such a back-end, we may detect such incompatibilities during virtualization (see Section 5.7). Some back-end systems (e.g., Lantern [38]) may require return types for recursive functions: Snek does not currently provide type inference for Rep types, but is able to generate type annotations for recursive functions if provided.

5.6 Introducing @stage

With the functionality now in place such that Snek enables users to perform multi-stage programming, we provide a new decorator, `@stage`. `@stage` allows for users to provide a list of `Rep` parameters and call a previously transformed function (i.e., decorated with `@lms`) easily. For example, given the program in Figure 10 (left), a user may add the the required staging function with a `Rep` parameter (Figure 10 (right)).

The `__init__` method defined in `@stage` immediately executes the body of the decorated function, generating the appropriate code wrapped within a `def` statement in the corresponding S-Expression. `@stage` may be provided with a hook into the downstream back-end, such that `@stage.__call__` will trigger the execution of the resultant code. We provide an example of this in Section 5.8.

[5] http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html

```
@lms
def aggregate(x):
  ret = 0
  while x > 0:
    ret = ret + x
    x = x - 1
  return ret
```

```
@stage
def stage_aggregate(x):
  return aggregate(x)
```

Figure 10. A sum function implemented in Python and decorated with `@lms` (left), and the corresponding staging call (right).

5.7 AutoGraph

```
def foo():
  # returns an integer
  x = my_int_fun()
   # returns a bool
  cond = my_fun()
  if cond: x = x + 1
  else: x = x - 1

with ag__.function_scope('foo'):
  x = my_int_fun()
  cond = my_fun()
  cond_1 = cond
  # continue on the right...
```

```
# ... here
def if_true():
  with ag__.function_scope('if_true'):
    x_1, = x,
    x_1 = x_1 + 1
    return x_1

def if_false():
  with ag__.function_scope('if_false'):
    x_2, = x,
    x_2 = x_2 - 1
    return x_2
x = ag__.if_stmt(cond_1, if_true, if_false)
```

Figure 11. Code which requires lifting in Snek (left), and code generated using AutoGraph (right).

We implement a production system targeting the TensorFlow runtime, which we dub AutoGraph.

Analysis Techniques. AutoGraph contains a number of analyses aimed at providing more specialized code generation. These include control flow graph construction between each transformation, activity analysis to track multiple assignments, liveness analysis, type and value analysis, and more [16]. Of particular interest is type and value analysis: this, combined with some decisions concerning our intermediate Python, enables AutoGraph to forgo the lifting transformations required in Section 3 (see Section 3.1 for implementation details), which simplifies the generated Python code. Consider the first example given in Section 3.1 (shown in Figure 11, left). Running this code in AutoGraph yields the intermediate Python code found in Figure 11, right. Of note here is the absence of any lifting constructs, while `x` is now assigned the result of the `if/else` statement (i.e., AutoGraph treats all virtualized operators as expressions). AutoGraph's `if_stmt` function returns values for all variables which are updated within the extracted functions: this can only be known through the use of the various analyses presented.

Back-end Code Generation. AutoGraph in its current incarnation is designed to be a front-end tool specifically used for easier interaction with TensorFlow [1]. TensorFlow programs consist almost entirely of API calls which generate computation graphs to be interpreted by the highly-optimized TensorFlow runtime. While TensorFlow is typically considered to be among the best performing machine learning back-ends for programs expressible in TensorFlow, the interface of API-centric programming poses difficult for many new users (especially regarding control flow, which also must be expressed through provided TensorFlow APIs [16]). However, with the ability to perform source code transformations and staging in the manner described in Snek, AutoGraph elects to generate code which will directly build a TensorFlow computation graph, while allowing users to program in idiomatic Python. Consider the following program which contains data-dependent control flow:

```
def square_if_positive(x): if x > 0: x = x * x  else: x = 0.0
  return x
```

This can be transformed and run using the following:

```
tf_square_if_positive = autograph.to_graph(square_if_positive)

with tf.Graph().as_default():
  g_out1 = tf_square_if_positive(tf.constant( 9.0))
  g_out2 = tf_square_if_positive(tf.constant(-9.0))
  with tf.Session() as sess:
    print('Graph results: %2.2f, %2.2f\n' \
      % (sess.run(g_out1), sess.run(g_out2)))
```

This produces the expected results of `81.00` and `0.00`[6].

However, these results are not computed in an eager fashion, though this is a possibility in TensorFlow [31]. Inspecting the generated TensorFlow graph shows the expected nodes, including all data-dependent control flow nodes.[7]

As stated in Section 5.5, AutoGraph is designed with the capability of verifying that input programs can be expressed in TensorFlow. Programs which contain recurrent functions are examples which will not pass this compatibility checking phase due to TensorFlow's lack of support for expressing in-graph functions [1, 16]. However, as shown in Snek, this limitation is solely on the part of TensorFlow: any modification which results in TensorFlow's ability to express in-graph computations will simply require implementing @`rep_fun` as shown in Section 5.6. Due to the benefits AutoGraph provides, as well as the required coupling with TensorFlow back-end properties, AutoGraph will be incorporated in the TensorFlow 2.0 release.

5.8 Integrating with Lightweight Modular Staging

In order to examine Snek's ability to interface with generative programming frameworks as downstream back-ends, we chose to utilize an existing parser from the LMS-Black project on GitHub [3] which is capable of taking S-Expression and generating Scala ASTs. With these ASTs in place, we are able to target the Lantern engine [38], which is implemented using Lightweight Modular Staging, as a back-end. We configure Lantern to utilize LMS's generative capabilities to produce C code, and use an off-the-shelf tool[8] to link to this generated code from Snek, thus allowing our @`stage` decorator (Section 5.6) to directly execute the final result. Given the program shown in Figure 6 (left), we add the required staging function decorated with @`stage` (Figure 12 (left)).

```
@stage
def stage_x(x):
  return run(x)
```

```
int x1(int x2) {
  int32_t x3 = x2 * x2;
  return x3;
}
```

Figure 12. Staging using @`stage` (left), and the generated C code from Lantern (right).

[6] https://www.tensorflow.org/guide/autograph

[7] We elide this graph for presentation.

[8] http://www.swig.org/

This results in the remainder of the code from Figure 6 being generated, with C code shown in Figure 12 generated using Lantern (bootstrapping code elided for simplicity). We can then call this from Python: `stage_x(10)` yields the expected result of `100`.

6 Staging in Snek: The Hard Parts

In this section, we present features of Python which may impact design decisions when implementing staging in the style of Lightweight Modular Staging. We also list limitations and detail how these limitations may be overcome either in future work, where applicable, or the workarounds currently (and perhaps permanently) in place within Snek.

6.1 Statements

Due to its functional nature, Scala (and therefore, Lightweight Modular Staging) contains no notion of statements (i.e., only expressions), even control flow structures like `if/else`. This allows users to use `if/else` expressions anywhere a normal expression would be used:

```
def aFunc(x: Int) = {  val a = if (x > 0) x; a }
```

Here, `aFunc` is of type `(Int ⇒ AnyVal)`, as `a` is of type `AnyVal`: the (implicit) branch in which the conditional does not hold will, by default, evaluate to the unit literal `()` to `a`. As such, the type of `a` must the closest common ancestor type to which both `x` and `()` conform: `AnyVal`. However, we can complicate things further through the use of a `return` in a single branch:

```
def aFunc2(x: Int): AnyVal = { val a = if (x > 0) return x else "str"; a }
```

Again, `aFunc2` is of type `(Int ⇒ AnyVal)`, but `a` is of type `String`, as the `return` keyword in Scala bypasses the default evaluation behavior in Scala which causes the enclosing *scope* to evaluate to the given value, instead causing the enclosing *function* to evaluate to the argument of return (i.e., `return` in Scala has side-effects).

In Python, however, `if/else` blocks are treated as statements, and do not evaluate to a value as in Scala. Indeed, there is no notion of "returning" a value to the enclosing scope, as all returns must be explicitly marked via the `return` keyword, returning a value from the enclosing *function.* This becomes of particular importance when we examine the staging of control flow structures.

Consider, for example, the following Python code:

```
def example_if(x):  if x >= 0: return x  else: print('negative')
```

If `x` is not a staged value, this function will either return `x` or `None`; as these values both exist at compile time, we encounter no problem. However, if we aim to stage this structure, Snek must know whether to propagate the return beyond the enclosing function through the use of a `NonLocalReturnValue` (Section 3.4), or if the value being returned should simply be staged (i.e., the result of reifying a function generated via a Snek transformation). With a function such as `example_if`, it seems trivial to simply move the "remainder" of the function (in this case, the implicit `return None` which exists after the `if/else`) into the `else` branch. However, one can imagine such an `if/else` statement contained within a loop:

```
def example_while(x):
  while x > 0:
    if x is 3: return x
    x = x - 1
  return x
```

While instances such as this may be resolved using various forms of analysis (including dataflow analysis or CPS-style transformations), Snek elects to simply impose the restriction upon users that return statements may not exist within loops, and `if/else` statements must either have a return statement in every branch, or contain no return statements. We note that this is consistent with the behavior currently exhibited by AutoGraph [16].

6.2 External Libraries.

A difficulty arises in dealing with function calls to external libraries to which we pass `Rep` values, such as using an existing PyTorch function call. In these instances, we use an overload of `Rep`'s `__getattr__` method to generate the library call.

6.3 Staging `while` Loops.

```
@lms
def staged_while(n):
  x = 0
  while x < n: x = x + 1
  return x
```

```
def staged_while(n):
  try:
    x = _var(); _assign(x, 0)
    def cond$1(): return (_read(x) < n)
    def body$1(): _assign(x, _read(x) + 1)
    _while(cond$1, body$1)
    _return(_read(x))
  except NonLocalReturnValue as r:
    return r.value
```

Figure 13. A `while` loop in Python before (left) and after (right) Snek transformations.

Consider the code in Figure 13. If `n` is a `Rep` value, this `while` loop should be present in the generated code. In order to determine the return type of `cond$1`, we must run this operation. However, running this operation will generate code if `n` is a `Rep` value. As such, Snek generates code for the conditional twice: once before the loop, and once in the correct location. Running `staged_while` with a `Rep` parameter thus yields S-Expression as follows:

```
(def staged_while ...
  (let x10 (< x9 in1) ... ; determining whether the loop is staged
  (while ... (let x12 (< x11 in1) x12)...))...) ; staging the conditional
```

6.4 Limitations

Ternary Operators. The use of Python's ternary operators are currently disallowed in Snek, due to the fact that $[\![\ ]\!]_v$ would require extracting multiple functions, but ternary operators must be a single expression.

`lambda` Functions. Using a `lambda` function in Snek without modification is permitted, and functions as one may expect: the function executes as normal for unstaged values, and generates code for `Rep` values based on the operations executed. However, due to Python's restriction that `lambda` functions may contain at most one statement, and may not contain `return` statements or assignments, the only control flow structures which may appear in a `lambda` are the ternary operators which are currently disallowed in Snek.

Snek also provides the ability to stage `lambda` functions. However, simply virtualizing `lambda` is insufficient, as Python does not provide syntax to differentiate between *calling* a `lambda` on a `Rep` value and *staging* the `lambda` itself. As such, users must explicitly call Snek's provided `stageLambda` on `lambda`s they wish to appear in the generated code.

Miscellaneous. Snek does not currently stage `yield` statements, `Exception`s, or `class` definitions, due to the inability of many back-ends to implement this functionality.

Table 1. Training 5 Epochs of MNIST Using Figure 14 With Different Front-End Systems

PyTorch	122.4
Snek + Pytorch, Unstaged	124
Snek + Pytorch, Staged	119.8
Torch Script	128.8
Snek + Torch Script	127.2

7 Evaluation

```
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
```

Figure 14. Simplified implementation of MNIST in PyTorch, with optional hyperparameter to specify activation function.

In this section, we assess the performance effects of both virtualization and staging in Snek and AutoGraph. We aim to quantify what overhead arises as a result of applying $[\![\ ]\!]_v$, comparing an implementation of the standard MNIST benchmark in PyTorch with the same model decorated with `@lms`. We compare these with a model generated via `@torch.jit.script`, as well as a model generated from Snek using `@stage` which is then reinterpreted as Torch Script code. Finally, we present an evaluation of AutoGraph on a simple, realistic reinforcement learning training task.

7.1 Generating PyTorch/Torch Script from Snek

To evaluate the performance benefits of staging in Snek, we implement a parser which generates Python code from S-Expression. To provide a more direct comparison with Torch Script, we elect to only generate the model, and leave all training code unmodified. We note that a number of specializations could occur (e.g., unrolling the training loop), but do not capture them in this evaluation.

7.2 Environment

All Snek experiments were 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 use Python 3.5.2, Scala 2.11.6, gcc 5.4.0, torch 0.4.1, and Ubuntu 16.04. All AutoGraph experiments were conducted on a single machine with one 6-core Intel Xeon E5-1650 CPU running at 3.60GHz, and 64 GB or RAM, using Python 2.7, and Debian 4.18.

7.3 MNIST Dataset

MNIST [15] is a standard introductory program for machine learning programmers to perform image classification. 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). For presentation, we elect to simplify the implementation provided by PyTorch [9], using a single fully-connected layer (consisting of two `Linear` layers). However, we also allow users to define which activation function they wish to use via a hyperparameter, yielding the implementation [10] found in Figure 14. We use this as the starting point for all of our implementations tested in Table 1, with Snek and Torch Script annotations requiring minor modifications.

7.4 MNIST in PyTorch

We train the model from Figure 14 using a naive PyTorch implementation, Snek without staging (just virtualization), and Torch Script, as well as the performance of staging in Snek on both the PyTorch and Torch Script implementations (utilizing the code generated from the parser described in Section 7.1). We target the PyTorch runtime as a back-end in all these experiments, and report the wall clock times (in seconds) of training for 5 epochs in Table 1 (average of 5 runs). We note that all observable behavior (e.g., training loss) appears identical between the implementations (except performance).

Virtualization Overhead. It is expected that Snek's functionality modulo staging will introduce some level of overhead. This is primarily due to the indirection introduced via our virtualization functions, including `isinstance` checks to determine whether to stage a particular operation. As seen in Table 1 (a) and (b), we see results as expected. However, the introduced overhead is limited to only a 1.30% average performance loss.

Staging Benefits with Snek. Due to the fact that nearly all operations in the implementation provided in Figure 14 are data- dependent, Snek is unable to "stage away" much in the generated code. Indeed, the only operation absent in the generated code is data-independent conditional. However, while in our application and in our environment there is little other specialization which can occur, in general even staging data-dependent operations may yield significant performance gains. Of particular note is the situation in which one wishes to access a tensor's value when that tensor lives on another device. By staging the relevant control flow constructs to target the appropriate device, this cost is mitigated. Indeed, even the simple removal of the single `if/else` branching statement in our implementation yields a 2.10% performance benefit on average.

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

[10]We elide all training details.

Torch Script Results. The results in Table 1 show a 5.20% performance degradation on average in the translated Torch Script model. However, when generating Torch Script from Snek, we find an average loss of only 3.90%, again due to Snek's ability to remove the data-independent control flow from the generated model. We note that Torch Script [22] is designed primarily for usability, with the main benefit being that users may generate graphs to be used at a later point, and is not yet in a production state.

7.5 Targeting Lantern

In order to evaluate using Snek with a statically-typed back-end, we elect to target the Lantern engine [38] as a machine learning back-end. We use the parser provided by Amin et al. [3] to convert the S-Expression generated by Snek into a Scala AST upon which Lantern may reason. Due to the fact that Lantern is built using Lightweight Modular Staging (LMS) [23], and that Scala is statically typed, all expressions must be given some type (possibly a `Rep[T]` type, for staged expressions). This differentiation alone allows for some staging opportunities which are impossible in Snek without further analysis, including no longer always requiring the lifting of variables which may be assigned multiple values (in LMS, it is impossible to assign a value of type `Rep[T]` to an identifier with type `T`).

With the ability to run the MNIST example shown in Figure 14 using Lantern as a back-end, we implement the same program in Lantern for comparison as a baseline. Indeed, the resultant performance is identical between the two implementations, as the added stage between Snek and Lantern enables us yet another stage for optimization (both implementations require 25.1 seconds on average). We note that in a production system, this code which interprets the Scala ASTs and translates the calls to machine learning kernels present in the front-end system into corresponding calls available in the back-end system would typically be implemented by a domain expert over the relevant back-end. As such, it is *not* intended that end users need to implement (or even know about) these back-end functions when designing from a higher level of abstraction.

7.6 AutoGraph

We evaluate the use of AutoGraph in a reinforcement learning (RL) benchmark. Applications in RL typically involve non-trivial control flow and book-keeping, as well as dispatch to a simulation environment. Specifically, we train a two-layer policy network on the cart-pole problem, using AutoGraph, and two equivalent unstaged implementations: one in TensorFlow Eager and another in PyTorch. The training procedure requires both data-dependent control flow (e.g. the episode loop) and data-independent control flow (e.g. iterating over the model parameters and gradients). To ensure identical work loads, we use a fake environment that ensures the episode length is kept constant. For the purpose of benchmarking, we disregard any learning and generate random

Table 2. Two-Layer Policy Network Training On Cart-Pole With Policy Gradients

Hidden Layer Size	10	100	1000
AutoGraph	0.39	0.40	0.46
TF Eager	0.91	0.93	1.06
PyTorch	0.56	0.58	0.68

observations and actions. We vary the size of the hidden layer in the network, while keeping the episode length fixed, providing random rewards. Each training step averages the gradients scaled by the cumulative discounted rewards over 20 forward plays. We report the wall clock time (in seconds) it takes to perform 10 steps of learning in Table 2.

While the source code is largely similar between all three implementations, the AutoGraph implementation shows speed improvements of 30-57% over the unstaged counterparts. This is expected due to the significant potential for optimization in the staged environment (the TensorFlow graph), as well as the elimination of data-independent control flow from the graph.

8 Related Work

Multi-stage Programming. Multi-stage programming is a well-studied area. Tools like Terra [9] showcase the ability to metaprogram to an intermediate language specifically designed to integrate between user-facing code and highly-optimized machine code. Of most relevance to Snek is Lightweight Modular Staging (LMS) [25], upon which Snek is based. LMS uses a specialized type annotation `Rep[T]` to allow users to mark values as being known at runtime, with all other values known at compile time, and relies on virtualization of Scala built-ins [17]. 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 [10, 26, 27, 29, 30, 40]. Of most relevance here is Lantern [38, 39], which uses LMS to provide a differentiable programming interface. Amin and Rompf [4] also showed how multi-stage programming can be used to reduce the overhead inherent in different interpreter boundaries.

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 [13]. Snek attempts to specialize based on user intent, with this intent expressed through the use of function decorators.

Metaprogramming for ML. A number of machine learning systems also apply source code transformations to Python in order to provide users an easier-to-use, high-level programming interface targeting specialized back-ends. PyTorch's Torch Script [22] provides users with `@torch.jit.trace` and `@torch.jit.script` in order to extract computation graphs from PyTorch code (which is designed to be as close to idiomatic Python as possible).

Other frameworks rely on source code transformations to accelerate machine learning targeting Python: Myia [35] converts Python to a differentiable programming language; Cython [5] translates Python functions into C where possible; Tangent [36] generates Python functions which calculate derivatives (i.e., automatic differentiation through source code transformations); Weld [19] aims to provide a universal IR for high-performance computing libraries in Python; Theano [2] constructs a computation graph and performs symbolic differentiation; and Keras [8] is built on Theano and provides an open source neural network library for users. Frameworks like Tensor Comprehensions [37] also exist, though these operate on mathematical descriptions of computation graphs, and do not directly transform Python in the manner described here.

The Open Neural Network eXchange (ONNX) [18] also allows users to extract computation graphs for later usage through the use of tracing. ONNX provides encoding and decoding APIs for a number of popular machine learning front-ends and back-ends, respectively. torch-autograd [34] and Chainer [33] also perform tracing, though neither focus primarily on the extraction of computation graphs.

Python Semantics. λ_π [21] is a formal grammar 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 and AutoGraph 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 [12], as well as in the K semantic framework [12, 28].

9 Conclusions

In this paper, we presented a virtualization rules which enable "staging based on types" in Python. This staging does not require any type annotations on the part of users, and in most cases, the code changes required are to annotate the desired functions with the `@lms` decorator. Virtualization in this fashion gives the full expressive power of Python as well as the efficiency of arbitrary back-ends through the use of back-end agnostic code generation strategies. These capabilities are provided in a prototype called Snek, as well as a production system targeting TensorFlow, called AutoGraph. In future work, we aim to increase the coverage of Python constructs which Snek may virtualize, as well as providing greater coverage of popular machine learning libraries and constructs. We also look to serve additional domains, rather than the current focus on machine learning libraries.

References

[1] 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.

[2] 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).

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

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

[5] 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

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

[7] 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!)*.

[8] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras.

[9] 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.

[10] 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.

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

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

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

[14] 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

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

[16] 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).

[17] 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)*.

[18] ONNX Contributors. 2018. Open Neural Network Exchange. https://github.com/onnx/onnx. Accessed: 2018-09-24.

[19] 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.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[21] 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

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

[23] 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

[24] 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

[25] 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.

[26] 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)*.

[27] 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.

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

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

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

[31] TensorFlow Contributors. 2018. Eager Execution. https://www.tensorflow.org/guide/eager. Accessed: 2018-10-31.

[32] TensorFlow Contributors. 2018, howpublished=https://www.tensorflow.org/swift/. Swift for TensorFlow. Accessed: 2018-10-31.

[33] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *NIPS 2015 LearningSys Workshop*, Vol. 5.

[34] Torch Autograd Contributors. 2018. torch-autograd. https://github.com/twitter/torch-autograd. Accessed: 2018-09-25.

[35] 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*.

[36] Bart van Merriënboer, Dan Moldovan, and Alexander B. Wiltschko. 2018. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. *CoRR* abs/1809.09569 (2018).

[37] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).

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

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

[40] Guannan Wei, James M. Decker, and Tiark Rompf. 2018. Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *PACMPL* 2, ICFP (2018), 105:1–105:28.