Autodidax: JAX core from scr

Contents

Ever want to learn how JAX works, but the implementation seemed impenetrable? Well, you're in luck! By reading this tutorial, you'll learn every big idea in JAX's core system. You'll even get clued into our weird jargon!

This is a work-in-progress draft. There are some important ingredients missing, still to come in parts 5 and 6 (and more?). There are also some simplifications here that we haven't yet applied to the main system, but we will.

Part 1: Transformations as interpreters: standard evaluation, `jvp`, and `vmap`

We want to transform functions that look like this:

```
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z
```

Think of functions like `sin` and the arithmetic operations underlying the infix operators (`mul`, `add`, and `neg`) as primitive operations, meaning atomic units of processing rather than compositions.

“Transform” means “interpret differently.” Instead of standard interpretation where we apply primitive operations to numerical inputs to produce numerical outputs, we want to override primitive application and let different values flow through our program. For example, we might want to replace the application of every primitive with an application of [its JVP rule](), and let primal-tangent pairs flow through our program. Moreover, we want to be able to compose multiple transformations, leading to stacks of interpreters.

JAX core machinery

We can implement stacks of interpreters and even have them all discharge on the fly as we execute the Python function to be transformed. To start, let’s define these primitives so that we can intercept their application:

```python
from typing import NamedTuple

class Primitive(NamedTuple):
  name: str

add_p = Primitive('add')
mul_p = Primitive('mul')
neg_p = Primitive("neg")
sin_p = Primitive("sin")
cos_p = Primitive("cos")
reduce_sum_p = Primitive("reduce_sum")
greater_p = Primitive("greater")
less_p = Primitive("less")
transpose_p = Primitive("transpose")
broadcast_p = Primitive("broadcast")

def add(x, y): return bind1(add_p, x, y)
def mul(x, y): return bind1(mul_p, x, y)
def neg(x): return bind1(neg_p, x)
def sin(x): return bind1(sin_p, x)
def cos(x): return bind1(cos_p, x)
def greater(x, y): return bind1(greater_p, x, y)
def less(x, y): return bind1(less_p, x, y)
def transpose(x, perm): return bind1(transpose_p, x, perm=perm)
def broadcast(x, shape, axes): return bind1(broadcast_p, x, shape=shape,
axes=axes)
def reduce_sum(x, axis=None):
  if axis is None:
    axis = tuple(range(np.ndim(x)))
  if type(axis) is int:
    axis = (axis,)
  return bind1(reduce_sum_p, x, axis=axis)

def bind1(prim, *args, **params):
  out, = bind(prim, *args, **params)
  return out
```

We’ll set up array data types and infix operator methods in a moment.

A `Primitive` is just an object with a name, to which we attach our interpretation rules (one for each transformation). The `bind` function is our interception point: it'll figure out which transformation rule to apply, based on how the arguments are boxed in tracers and what interpreters are active.

The functions that user code calls, like `add` and `sin`, are just wrappers around calls to `bind`. These wrappers let us control how arguments are passed to `bind`, and in particular we follow a handy internal convention: when we call `bind`, we pass values representing array data as positional arguments, and we pass metadata like the `axis` argument to `sum_p` via keyword. This calling convention simplifies some core logic (since e.g. instances of the `Tracer` class to be defined below can only occur in positional arguments to `bind`). The wrappers can also provide docstrings!

We represent active interpreters as a stack. The stack is just a simple `list`, and each element is a container with an integer level (corresponding to the element's height in the stack), an interpreter type (which we'll call a `trace_type`), and an optional field for any global data the interpreter needs. We call each element a `MainTrace`, though maybe "Interpreter" would be more descriptive.

```
from contextlib import contextmanager
from typing import Type, List, Tuple, Sequence, Optional, Any

class MainTrace(NamedTuple):
  level: int
  trace_type: Type['Trace']
  global_data: Optional[Any]

trace_stack: List[MainTrace] = []
dynamic_trace: Optional[MainTrace] = None  # to be employed in Part 3

@contextmanager
def new_main(trace_type: Type['Trace'], global_data=None):
  level = len(trace_stack)
  main = MainTrace(level, trace_type, global_data)
  trace_stack.append(main)

  try:
    yield main
  finally:
    trace_stack.pop()
```

When we're about to apply a transformation, we'll push another interpreter onto the stack using `new_main`. Then, as we apply primitives in the function, we can think of the `bind` first being interpreted by the trace at the top of the stack (i.e. with the highest level). If that first interpreter itself binds other primitives in its interpretation rule for the primitive, like how the JVP rule of `sin_p` might bind `cos_p` and `mul_p`, then those `bind` calls will be handled by the interpreter at the next level down.

What goes at the bottom of the interpreter stack? At the bottom, we know all the transformation interpreters are finished, and we just want to do standard evaluation. So at the bottom we'll put an evaluation interpreter.

Let's sketch out the interface for interpreters, which is based on the `Trace` and `Tracer` base classes. A `Tracer` represents a boxed-up value, perhaps carrying some extra context data used by the interpreter. A `Trace` handles boxing up values into `Tracers` and also handles primitive application.

```
class Trace:
  main: MainTrace

  def __init__(self, main: MainTrace) -> None:
    self.main = main

  def pure(self, val): assert False  # must override
  def lift(self, val): assert False  # must override

  def process_primitive(self, primitive, tracers, params):
    assert False  # must override
```

The first two methods are about boxing up values in `Tracer`s, which are the objects that flow through the Python programs we transform. The last method is the callback we'll use to interpret primitive application.

The `Trace` itself doesn't contain any data, other than a reference to its corresponding `MainTrace` instance. In fact, multiple instances of a `Trace` might be created and discarded during an application of a transformation, whereas only a single `MainTrace` instance is created per application of a transformation.

As for `Tracer`s themselves, each one carries an abstract value (and forwards infix operators to it), and the rest is up to the transformation. (The relationship between `Tracer`s and `AbstractValue`s is that there's one `Tracer` per transformation, and at least one `AbstractValue` per base type, like arrays.)

```python
import numpy as np

class Tracer:
  _trace: Trace

  __array_priority__ = 1000

  @property
  def aval(self):
    assert False  # must override

  def full_lower(self):
    return self  # default implementation

  def __neg__(self): return self.aval._neg(self)
  def __add__(self, other): return self.aval._add(self, other)
  def __radd__(self, other): return self.aval._radd(self, other)
  def __mul__(self, other): return self.aval._mul(self, other)
  def __rmul__(self, other): return self.aval._rmul(self, other)
  def __gt__(self, other): return self.aval._gt(self, other)
  def __lt__(self, other): return self.aval._lt(self, other)
  def __bool__(self): return self.aval._bool(self)
  def __nonzero__(self): return self.aval._nonzero(self)

  def __getattr__(self, name):
    try:
      return getattr(self.aval, name)
    except AttributeError:
      raise AttributeError(f"{self.__class__.__name__} has no attribute
{name}")

def swap(f): return lambda x, y: f(y, x)
```

```python
class ShapedArray:
  array_abstraction_level = 1
  shape: Tuple[int, ...]
  dtype: np.dtype

  def __init__(self, shape, dtype):
    self.shape = shape
    self.dtype = dtype

  @property
  def ndim(self):
    return len(self.shape)

  _neg = staticmethod(neg)
  _add = staticmethod(add)
  _radd = staticmethod(swap(add))
  _mul = staticmethod(mul)
  _rmul = staticmethod(swap(mul))
  _gt = staticmethod(greater)
  _lt = staticmethod(less)

  @staticmethod
  def _bool(tracer):
    raise Exception("ShapedArray can't be unambiguously converted to bool")

  @staticmethod
  def _nonzero(tracer):
    raise Exception("ShapedArray can't be unambiguously converted to bool")

  def str_short(self):
    return f'{self.dtype.name}[{",".join(str(d) for d in self.shape)}]'

  def __hash__(self):
    return hash((self.shape, self.dtype))

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.shape == other.shape and self.dtype == other.dtype)

  def __repr__(self):
    return f"ShapedArray(shape={self.shape}, dtype={self.dtype})"

class ConcreteArray(ShapedArray):
  array_abstraction_level = 2
  val: np.ndarray

  def __init__(self, val):
    self.val = val
    self.shape = val.shape
    self.dtype = val.dtype

  @staticmethod
  def _bool(tracer):
    return bool(tracer.aval.val)

  @staticmethod
  def _nonzero(tracer):
    return bool(tracer.aval.val)

def get_aval(x):
  if isinstance(x, Tracer):
```

```
    return x.aval
  elif type(x) in jax_types:
    return ConcreteArray(np.asarray(x))
  else:
    raise TypeError(x)

jax_types = {bool, int, float,
             np.bool_, np.int32, np.int64, np.float32, np.float64,
np.ndarray}
```

Notice that we actually have two `AbstractValue`s for arrays, representing different levels of abstraction. A `ShapedArray` represents the set of all possible arrays with a given shape and dtype. A `ConcreteArray` represents a singleton set consisting of a single array value.

Now that we've set up the interpreter stack, the Trace/Tracer API for interpreters, and abstract values, we can come back to implement `bind`:

```
def bind(prim, *args, **params):
  top_trace = find_top_trace(args)
  tracers = [full_raise(top_trace, arg) for arg in args]
  outs = top_trace.process_primitive(prim, tracers, params)
  return [full_lower(out) for out in outs]
```

The main action is that we call `find_top_trace` to figure out which interpreter should handle this primitive application. We then call that top trace's `process_primitive` so that the trace can apply its interpretation rule. The calls to `full_raise` just ensure that the inputs are boxed in the top trace's `Tracer` instances, and the call to `full_lower` is an optional optimization so that we unbox values out of `Tracer`s as much as possible.

```
import operator as op

def find_top_trace(xs) -> Trace:
  top_main = max((x._trace.main for x in xs if isinstance(x, Tracer)),
                 default=trace_stack[0], key=op.attrgetter('level'))
  if dynamic_trace and dynamic_trace.level > top_main.level:
    top_main = dynamic_trace
  return top_main.trace_type(top_main)
```

In words, ignoring the `dynamic_trace` step until Part 3, `find_top_trace` returns the highest-level interpreter associated with the `Tracer`s on its inputs, and otherwise returns the interpreter at the bottom of the stack (which is always an evaluation trace, at least for now). This is a deviation from the description above, where we always start by running the interpreter at the top of the stack and then work our way down, applying every interpreter in the stack. Instead, we're only applying an interpreter when the input arguments to a primitive bind are boxed in a `Tracer` corresponding to that interpreter. This optimization lets us skip irrelevant transformations, but bakes in an assumption that transformations mostly follow data dependence (except for the special bottom-of-the-stack interpreter, which interprets everything).

An alternative would be to have every interpreter in the stack interpret every operation. That's worth exploring! JAX is designed around data dependence in large part because that's so natural for automatic differentiation, and JAX's roots are in autodiff. But it may be over-fit.

```
def full_lower(val: Any):
  if isinstance(val, Tracer):
    return val.full_lower()
  else:
    return val

def full_raise(trace: Trace, val: Any) -> Tracer:
  if not isinstance(val, Tracer):
    assert type(val) in jax_types
    return trace.pure(val)
  level = trace.main.level
  if val._trace.main is trace.main:
    return val
  elif val._trace.main.level < level:
    return trace.lift(val)
  elif val._trace.main.level > level:
    raise Exception(f"Can't lift level {val._trace.main.level} to {level}.")
  else:  # val._trace.level == level
    raise Exception(f"Different traces at same level: {val._trace},
{trace}.")
```

The logic in `full_raise` serves to box values into `Tracer`s for a particular `Trace`, calling different methods on the `Trace` based on context: `Trace.pure` is called on non-`Tracer` constants, and `Trace.lift` is called for values that are already `Tracer`s from a lower-level interpreter. These two methods could share the same implementation, but by distinguishing them in the core logic we can provide more information to the `Trace` subclass.

That's it for the JAX core! Now we can start adding interpreters.

Evaluation interpreter

We'll start with the simplest interpreter: the evaluation interpreter that will sit at the bottom of the interpreter stack.

```
class EvalTrace(Trace):
  pure = lift = lambda self, x: x  # no boxing in Tracers needed

  def process_primitive(self, primitive, tracers, params):
    return impl_rules[primitive](*tracers, **params)

trace_stack.append(MainTrace(0, EvalTrace, None))  # special bottom of the
stack

# NB: in JAX, instead of a dict we attach impl rules to the Primitive instance
impl_rules = {}

impl_rules[add_p] = lambda x, y: [np.add(x, y)]
impl_rules[mul_p] = lambda x, y: [np.multiply(x, y)]
impl_rules[neg_p] = lambda x: [np.negative(x)]
impl_rules[sin_p] = lambda x: [np.sin(x)]
impl_rules[cos_p] = lambda x: [np.cos(x)]
impl_rules[reduce_sum_p] = lambda x, *, axis: [np.sum(x, axis)]
impl_rules[greater_p] = lambda x, y: [np.greater(x, y)]
impl_rules[less_p] = lambda x, y: [np.less(x, y)]
impl_rules[transpose_p] = lambda x, *, perm: [np.transpose(x, perm)]

def broadcast_impl(x, *, shape, axes):
  for axis in sorted(axes):
    x = np.expand_dims(x, axis)
  return [np.broadcast_to(x, shape)]
impl_rules[broadcast_p] = broadcast_impl
```

With this interpreter, we can evaluate user functions:

```
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z

print(f(3.0))
```

```
2.7177599838802657
```

Woo! Like going around in a big circle. But the point of this indirection is that now we can add some real transformations.

Forward-mode autodiff with `jvp`

First, a few helper functions:

```python
def zeros_like(val):
  aval = get_aval(val)
  return np.zeros(aval.shape, aval.dtype)

def unzip2(pairs):
  lst1, lst2 = [], []
  for x1, x2 in pairs:
    lst1.append(x1)
    lst2.append(x2)
  return lst1, lst2

map_ = map
def map(f, *xs):
  return list(map_(f, *xs))

zip_ = zip
def zip(*args):
  fst, *rest = args = map(list, args)
  n = len(fst)
  for arg in rest:
    assert len(arg) == n
  return list(zip_(*args))
```

The `Tracer` for forward-mode autodiff carries a primal-tangent pair. The `Trace` applies JVP rules.

```python
class JVPTracer(Tracer):
  def __init__(self, trace, primal, tangent):
    self._trace = trace
    self.primal = primal
    self.tangent = tangent

  @property
  def aval(self):
    return get_aval(self.primal)

class JVPTrace(Trace):
  pure = lift = lambda self, val: JVPTracer(self, val, zeros_like(val))

  def process_primitive(self, primitive, tracers, params):
    primals_in, tangents_in = unzip2((t.primal, t.tangent) for t in tracers)
    jvp_rule = jvp_rules[primitive]
    primal_outs, tangent_outs = jvp_rule(primals_in, tangents_in, **params)
    return [JVPTracer(self, x, t) for x, t in zip(primal_outs, tangent_outs)]

jvp_rules = {}
```

Notice both `pure` and `lift` package a value into a `JVPTracer` with the minimal amount of context, which is a zero tangent value.

Let’s add some JVP rules for primitives:

```python
def add_jvp(primals, tangents):
  (x, y), (x_dot, y_dot) = primals, tangents
  return [x + y], [x_dot + y_dot]
jvp_rules[add_p] = add_jvp

def mul_jvp(primals, tangents):
  (x, y), (x_dot, y_dot) = primals, tangents
  return [x * y], [x_dot * y + x * y_dot]
jvp_rules[mul_p] = mul_jvp

def sin_jvp(primals, tangents):
  (x,), (x_dot,) = primals, tangents
  return [sin(x)], [cos(x) * x_dot]
jvp_rules[sin_p] = sin_jvp

def cos_jvp(primals, tangents):
  (x,), (x_dot,) = primals, tangents
  return [cos(x)], [-sin(x) * x_dot]
jvp_rules[cos_p] = cos_jvp

def neg_jvp(primals, tangents):
  (x,), (x_dot,) = primals, tangents
  return [neg(x)], [neg(x_dot)]
jvp_rules[neg_p] = neg_jvp

def reduce_sum_jvp(primals, tangents, *, axis):
  (x,), (x_dot,) = primals, tangents
  return [reduce_sum(x, axis)], [reduce_sum(x_dot, axis)]
jvp_rules[reduce_sum_p] = reduce_sum_jvp

def greater_jvp(primals, tangents):
  (x, y), _ = primals, tangents
  out_primal = greater(x, y)
  return [out_primal], [zeros_like(out_primal)]
jvp_rules[greater_p] = greater_jvp

def less_jvp(primals, tangents):
  (x, y), _ = primals, tangents
  out_primal = less(x, y)
  return [out_primal], [zeros_like(out_primal)]
jvp_rules[less_p] = less_jvp
```

Finally, we add a transformation API to kick off the trace:

```python
def jvp_v1(f, primals, tangents):
  with new_main(JVPTrace) as main:
    trace = JVPTrace(main)
    tracers_in = [JVPTracer(trace, x, t) for x, t in zip(primals, tangents)]
    out = f(*tracers_in)
    tracer_out = full_raise(trace, out)
    primal_out, tangent_out = tracer_out.primal, tracer_out.tangent
  return primal_out, tangent_out
```

And with that, we can differentiate!

```python
x = 3.0
y, sin_deriv_at_3 = jvp_v1(sin, (x,), (1.0,))
print(sin_deriv_at_3)
print(cos(3.0))
```

```
-0.9899924966004454
-0.9899924966004454
```

```python
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z

x, xdot = 3., 1.
y, ydot = jvp_v1(f, (x,), (xdot,))
print(y)
print(ydot)
```

```
2.7177599838802657
2.979984993200891
```

```python
def deriv(f):
  return lambda x: jvp_v1(f, (x,), (1.,))[1]

print(deriv(sin)(3.))
print(deriv(deriv(sin))(3.))
print(deriv(deriv(deriv(sin)))(3.))
print(deriv(deriv(deriv(deriv(sin))))(3.))
```

```
-0.9899924966004454
-0.1411200080598672
0.9899924966004454
0.1411200080598672
```

```python
def f(x):
  if x > 0.:  # Python control flow
    return 2. * x
  else:
    return x

print(deriv(f)(3.))
print(deriv(f)(-3.))
```

```
2.0
1.0
```

Pytrees and flattening user functions’ inputs

and outputs

A limitation with `jvp_v1` is that it assumes the user function accepts arrays as positional arguments and produces a single array as output. What if it produced a list as output? Or accepted nested containers as inputs? It would be a pain to deal with all the possible containers in inputs and outputs at every layer of the stack. Instead, we can wrap the user function so that the wrapped version accepts arrays as inputs and returns a flat list of arrays as output. The wrapper just needs to unflatten its input, call the user function, and flatten the output.

Here's how we'd like to write `jvp`, assuming the user always gives us functions that take arrays as inputs and produces a flat list of arrays as outputs:

```
def jvp_flat(f, primals, tangents):
  with new_main(JVPTrace) as main:
    trace = JVPTrace(main)
    tracers_in = [JVPTracer(trace, x, t) for x, t in zip(primals, tangents)]
    outs = f(*tracers_in)
    tracers_out = [full_raise(trace, out) for out in outs]
    primals_out, tangents_out = unzip2((t.primal, t.tangent) for t in
tracers_out)
  return primals_out, tangents_out
```

To support user functions that have arbitrary containers in the inputs and outputs, here's how we'd write the user-facing `jvp` wrapper:

```
def jvp(f, primals, tangents):
  primals_flat, in_tree = tree_flatten(primals)
  tangents_flat, in_tree2 = tree_flatten(tangents)
  if in_tree != in_tree2: raise TypeError
  f, out_tree = flatten_fun(f, in_tree)
  primals_out_flat, tangents_out_flat = jvp_flat(f, primals_flat,
tangents_flat)
  primals_out = tree_unflatten(out_tree(), primals_out_flat)
  tangents_out = tree_unflatten(out_tree(), tangents_out_flat)
  return primals_out, tangents_out
```

Notice that we had to plumb the tree structure of the user function output back to the caller of `flatten_fun`. That information isn't available until we actually run the user function, so `flatten_fun` just returns a reference to a mutable cell, represented as a thunk. These side-effects are safe because we always run the user function exactly once. (This safe regime is the reason for the "linear" name in `linear_util.py`, in the sense of [linear types](https://en.wikipedia.org/wiki/Substructural_type_system).)

All that remains is to write `tree_flatten`, `tree_unflatten`, and `flatten_fun`.

▶ Show code cell source

▶ Show code cell source

With this pytree-handling `jvp` implementation, we can now handle arbitrary input and output containers. That'll come in handy with future transformations too!

```
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return {'hi': z, 'there': [x, y]}

x, xdot = 3., 1.
y, ydot = jvp(f, (x,), (xdot,))
print(y)
print(ydot)
```

```
{'hi': 2.7177599838802657, 'there': [3.0, 0.2822400161197344]}
{'hi': 2.979984993200891, 'there': [1.0, -1.9799849932008908]}
```

Vectorized batching with `vmap`

First, a couple helper functions, one for producing mapped abstract values from unmapped ones (by removing an axis), and one for moving batch dimensions around:

```
def mapped_aval(batch_dim, aval):
  shape = list(aval.shape)
  del shape[batch_dim]
  return ShapedArray(tuple(shape), aval.dtype)

def move_batch_axis(axis_size, src, dst, x):
  if src is not_mapped:
    target_shape = list(np.shape(x))
    target_shape.insert(dst, axis_size)
    return broadcast(x, target_shape, [dst])
  elif src == dst:
    return x
  else:
    return moveaxis(x, src, dst)

def moveaxis(x, src: int, dst: int):
  perm = [i for i in range(np.ndim(x)) if i != src]
  perm.insert(dst, src)
  return transpose(x, perm)
```

The `Tracer` for vectorized batching carries a batched value and an optional integer indicating which axis (if any) is the batch axis.

```python
from typing import Union

class NotMapped: pass
not_mapped = NotMapped()

BatchAxis = Union[NotMapped, int]

class BatchTracer(Tracer):
  def __init__(self, trace, val, batch_dim: BatchAxis):
    self._trace = trace
    self.val = val
    self.batch_dim = batch_dim

  @property
  def aval(self):
    if self.batch_dim is not_mapped:
      return get_aval(self.val)
    else:
      return mapped_aval(self.batch_dim, get_aval(self.val))

  def full_lower(self):
    if self.batch_dim is not_mapped:
      return full_lower(self.val)
    else:
      return self

class BatchTrace(Trace):
  pure = lift = lambda self, val: BatchTracer(self, val, not_mapped)

  def process_primitive(self, primitive, tracers, params):
    vals_in, bdims_in = unzip2((t.val, t.batch_dim) for t in tracers)
    vmap_rule = vmap_rules[primitive]
    val_outs, bdim_outs = vmap_rule(self.axis_size, vals_in, bdims_in,
**params)
    return [BatchTracer(self, x, bd) for x, bd in zip(val_outs, bdim_outs)]

  @property
  def axis_size(self):
    return self.main.global_data

vmap_rules = {}
```

Here we've implemented the optional `Tracer.full_lower` method, which lets us peel off a batching tracer if it's not needed because it doesn't represent a batched value.

For `BatchTrace`, analogous to `JVPTrace`, the methods `pure` and `lift` just box a value in a `BatchTracer` with the minimal amount of context, which in this case is a `batch_dim` taking the sentinel value `not_mapped`. Notice we use the `MainTrace`'s interpreter-global data field to store the batch axis size.

Next we can define batching interpreter rules for each primitive:

```python
from functools import partial

def binop_batching_rule(op, axis_size, vals_in, dims_in):
  (x, y), (x_bdim, y_bdim) = vals_in, dims_in
  if x_bdim != y_bdim:
    if x_bdim is not_mapped:
      x = move_batch_axis(axis_size, x_bdim, y_bdim, x)
      x_bdim = y_bdim
    else:
      y = move_batch_axis(axis_size, y_bdim, x_bdim, y)
  return [op(x, y)], [x_bdim]
vmap_rules[add_p] = partial(binop_batching_rule, add)
vmap_rules[mul_p] = partial(binop_batching_rule, mul)

def vectorized_unop_batching_rule(op, axis_size, vals_in, dims_in):
  (x,), (x_bdim,) = vals_in, dims_in
  return [op(x)], [x_bdim]
vmap_rules[sin_p] = partial(vectorized_unop_batching_rule, sin)
vmap_rules[cos_p] = partial(vectorized_unop_batching_rule, cos)
vmap_rules[neg_p] = partial(vectorized_unop_batching_rule, neg)

def reduce_sum_batching_rule(axis_size, vals_in, dims_in, *, axis):
  (x,), (x_bdim,) = vals_in, dims_in
  new_axis = tuple(ax + (x_bdim <= ax) for ax in axis)
  out_bdim = x_bdim - sum(ax < x_bdim for ax in axis)
  return [reduce_sum(x, new_axis)], [out_bdim]
vmap_rules[reduce_sum_p] = reduce_sum_batching_rule
```

Finally, we add a transformation API to kick off the trace:

```python
def vmap_flat(f, in_axes, *args):
  axis_size, = {x.shape[ax] for x, ax in zip(args, in_axes)
                if ax is not not_mapped}
  with new_main(BatchTrace, axis_size) as main:
    trace = BatchTrace(main)
    tracers_in = [BatchTracer(trace, x, ax) if ax is not None else x
                  for x, ax in zip(args, in_axes)]
    outs = f(*tracers_in)
    tracers_out = [full_raise(trace, out) for out in outs]
    vals_out, bdims_out = unzip2((t.val, t.batch_dim) for t in tracers_out)
  outs_transposed = [move_batch_axis(axis_size, bdim, 0, val_out)
                     for val_out, bdim in zip(vals_out, bdims_out)]
  return outs_transposed

def vmap(f, in_axes):
  def batched_f(*args):
    args_flat, in_tree = tree_flatten(args)
    in_axes_flat, in_tree2 = tree_flatten(in_axes)
    if in_tree != in_tree2: raise TypeError
    f_flat, out_tree = flatten_fun(f, in_tree)
    outs_flat = vmap_flat(f_flat, in_axes_flat, *args_flat)
    return tree_unflatten(out_tree(), outs_flat)
  return batched_f
```

```
def add_one_to_a_scalar(scalar):
  assert np.ndim(scalar) == 0
  return 1 + scalar

vector_in = np.arange(3.)
vector_out = vmap(add_one_to_a_scalar, (0,))(vector_in)

print(vector_in)
print(vector_out)
```

```
[0. 1. 2.]
[1. 2. 3.]
```

```
def jacfwd(f, x):
  pushfwd = lambda v: jvp(f, (x,), (v,))[1]
  vecs_in = np.eye(np.size(x)).reshape(np.shape(x) * 2)
  return vmap(pushfwd, (0,))(vecs_in)

def f(x):
  return sin(x)

jacfwd(f, np.arange(3.))
```

```
array([[ 1.        ,  0.        , -0.        ],
       [ 0.        ,  0.54030231, -0.        ],
       [ 0.        ,  0.        , -0.41614684]])
```

That's it for `jvp` and `vmap`!

Part 2: Jaxprs

The next transformations on the horizon are `jit` for just-in-time compilation and `vjp` for reverse-mode autodiff. (`grad` is just a small wrapper around `vjp`.) Whereas `jvp` and `vmap` only needed each `Tracer` to carry a little bit of extra context, for both `jit` and `vjp` we need much richer context: we need to represent *programs*. That is, we need jaxprs!

Jaxprs are JAX's internal intermediate representation of programs. They are explicitly typed, functional, first-order, and in ANF form. We need a program representation for `jit` because the purpose of `jit` is to stage computation out of Python. For any computation we want to stage out, we need to be able to represent it as data, and build it up as we trace a Python function. Similarly, `vjp` needs a way to represent the computation for the backward pass of reverse-mode autodiff. We use the same jaxpr program representation for both needs.

(Building a program representation is the most [free](#) kind of trace-transformation, and so except for issues around handling native Python control flow, any transformation could be implemented by first tracing to a jaxpr and then interpreting the jaxpr.)

Jaxpr data structures

The jaxpr term syntax is roughly:

```
jaxpr ::=
  { lambda <binder> , ... .
    let <eqn>
        ...
    in ( <atom> , ... ) }

binder ::= <var>:<array_type>
var ::= a | b | c | ...
atom ::= <var> | <literal>
literal ::= <int32> | <int64> | <float32> | <float64>

eqn ::= <binder> , ... = <primitive> [ <params> ] <atom> , ...
```

The syntax of types is:

```
jaxpr_type ::= [ <array_type> , ... ] -> [ <array_type> , ... ]
array_type ::= <dtype>[<shape>]
dtype ::= f32 | f64 | i32 | i64
shape ::= <int> , ...
```

How do we represent these as Python data structures? We reuse ShapedArrays to represent types, and we can represent the term syntax with a few Python structs:

```python
from typing import Set

class Var:
  aval: ShapedArray
  def __init__(self, aval): self.aval = aval

class Lit:
  val: Any
  aval: ShapedArray

  def __init__(self, val):
    self.aval = aval = raise_to_shaped(get_aval(val))
    self.val = np.array(val, aval.dtype)

Atom = Union[Var, Lit]

class JaxprEqn(NamedTuple):
  primitive: Primitive
  inputs: List[Atom]
  params: Dict[str, Any]
  out_binders: List[Var]

class Jaxpr(NamedTuple):
  in_binders: List[Var]
  eqns: List[JaxprEqn]
  outs: List[Atom]

  def __hash__(self): return id(self)
  __eq__ = op.is_

def raise_to_shaped(aval):
  return ShapedArray(aval.shape, aval.dtype)
```

Type-checking a jaxpr involves checking that there are no unbound variables, that variables are only bound once, and that for each equation the type of the primitive application matches the type of the output binders.

```python
class JaxprType(NamedTuple):
  in_types:  List[ShapedArray]
  out_types: List[ShapedArray]

  def __repr__(self):
    in_types = ', '.join(aval.str_short() for aval in self.in_types)
    out_types = ', '.join(aval.str_short() for aval in self.out_types)
    return f'({in_types}) -> ({out_types})'

def typecheck_jaxpr(jaxpr: Jaxpr) -> JaxprType:
  env: Set[Var] = set()

  for v in jaxpr.in_binders:
    if v in env: raise TypeError
    env.add(v)

  for eqn in jaxpr.eqns:
    in_types = [typecheck_atom(env, x) for x in eqn.inputs]
    out_types = abstract_eval_rules[eqn.primitive](*in_types, **eqn.params)
    for out_binder, out_type in zip(eqn.out_binders, out_types):
      if not out_type == out_binder.aval: raise TypeError
    for out_binder in eqn.out_binders:
      if out_binder in env: raise TypeError
      env.add(out_binder)

  in_types = [v.aval for v in jaxpr.in_binders]
  out_types = [typecheck_atom(env, x) for x in jaxpr.outs]
  return JaxprType(in_types, out_types)

def typecheck_atom(env: Set[Var], x: Atom) -> ShapedArray:
  if isinstance(x, Var):
    if x not in env: raise TypeError("unbound variable")
    return x.aval
  elif isinstance(x, Lit):
    return raise_to_shaped(get_aval(x.val))
  else:
    assert False
```

We can apply the function represented by a jaxpr to arguments with a simple interpreter.

```python
def eval_jaxpr(jaxpr: Jaxpr, args: List[Any]) -> List[Any]:
  env: Dict[Var, Any] = {}

  def read(x: Atom) -> Any:
    return env[x] if type(x) is Var else x.val

  def write(v: Var, val: Any) -> None:
    assert v not in env  # single-assignment
    env[v] = val

  map(write, jaxpr.in_binders, args)
  for eqn in jaxpr.eqns:
    in_vals = map(read, eqn.inputs)
    outs = bind(eqn.primitive, *in_vals, **eqn.params)
    map(write, eqn.out_binders, outs)
  return map(read, jaxpr.outs)

def jaxpr_as_fun(jaxpr: Jaxpr):
  return lambda *args: eval_jaxpr(jaxpr, args)
```

By using `bind` in the interpreter, this interpreter itself is traceable.

Building jaxprs with tracing

Now that we have jaxprs as a data structure, we need ways to produce these from tracing Python code. In general there are two variants of how we trace to a jaxpr; `jit` uses one and `vjp` uses the other. We'll start with the one used by `jit`, which is also used by control flow primitives like `lax.cond`, `lax.while_loop`, and `lax.scan`.

```python
def split_list(lst: List[Any], n: int) -> Tuple[List[Any], List[Any]]:
  assert 0 <= n <= len(lst)
  return lst[:n], lst[n:]

def partition_list(bs: List[bool], l: List[Any]) -> Tuple[List[Any],
List[Any]]:
  assert len(bs) == len(l)
  lists = lst1, lst2 = [], []
  for b, x in zip(bs, l):
    lists[b].append(x)
  return lst1, lst2
```

```python
# NB: the analogous class in JAX is called 'DynamicJaxprTracer'
class JaxprTracer(Tracer):
  __slots__ = ['aval']
  aval: ShapedArray

  def __init__(self, trace, aval):
    self._trace = trace
    self.aval = aval

# NB: the analogous class in JAX is called 'DynamicJaxprTrace'
class JaxprTrace(Trace):
  def new_arg(self, aval: ShapedArray) -> JaxprTracer:
    aval = raise_to_shaped(aval)
    tracer = self.builder.new_tracer(self, aval)
    self.builder.tracer_to_var[id(tracer)] = Var(aval)
    return tracer

  def get_or_make_const_tracer(self, val: Any) -> JaxprTracer:
    tracer = self.builder.const_tracers.get(id(val))
    if tracer is None:
      tracer = self.builder.new_tracer(self, raise_to_shaped(get_aval(val)))
      self.builder.add_const(tracer, val)
    return tracer
  pure = lift = get_or_make_const_tracer

  def process_primitive(self, primitive, tracers, params):
    avals_in = [t.aval for t in tracers]
    avals_out = abstract_eval_rules[primitive](*avals_in, **params)
    out_tracers = [self.builder.new_tracer(self, a) for a in avals_out]
    inputs = [self.builder.getvar(t) for t in tracers]
    outvars = [self.builder.add_var(t) for t in out_tracers]
    self.builder.add_eqn(JaxprEqn(primitive, inputs, params, outvars))
    return out_tracers

  @property
  def builder(self):
    return self.main.global_data

# NB: in JAX, we instead attach abstract eval rules to Primitive instances
abstract_eval_rules = {}
```

Notice that we keep as interpreter-global data a builder object, which keeps track of variables, constants, and eqns as we build up the jaxpr.

```python
class JaxprBuilder:
  eqns: List[JaxprEqn]
  tracer_to_var: Dict[int, Var]
  const_tracers: Dict[int, JaxprTracer]
  constvals: Dict[Var, Any]
  tracers: List[JaxprTracer]

  def __init__(self):
    self.eqns = []
    self.tracer_to_var = {}
    self.const_tracers = {}
    self.constvals = {}
    self.tracers = []

  def new_tracer(self, trace: JaxprTrace, aval: ShapedArray) -> JaxprTracer:
    tracer = JaxprTracer(trace, aval)
    self.tracers.append(tracer)
    return tracer

  def add_eqn(self, eqn: JaxprEqn) -> None:
    self.eqns.append(eqn)

  def add_var(self, tracer: JaxprTracer) -> Var:
    assert id(tracer) not in self.tracer_to_var
    var = self.tracer_to_var[id(tracer)] = Var(tracer.aval)
    return var

  def getvar(self, tracer: JaxprTracer) -> Var:
    var = self.tracer_to_var.get(id(tracer))
    assert var is not None
    return var

  def add_const(self, tracer: JaxprTracer, val: Any) -> Var:
    var = self.add_var(tracer)
    self.const_tracers[id(val)] = tracer
    self.constvals[var] = val
    return var

  def build(self, in_tracers: List[JaxprTracer], out_tracers:
List[JaxprTracer]
            ) -> Tuple[Jaxpr, List[Any]]:
    constvars, constvals = unzip2(self.constvals.items())
    t2v = lambda t: self.tracer_to_var[id(t)]
    in_binders = constvars + [t2v(t) for t in in_tracers]
    out_vars = [t2v(t) for t in out_tracers]
    jaxpr = Jaxpr(in_binders, self.eqns, out_vars)
    typecheck_jaxpr(jaxpr)
    jaxpr, constvals = _inline_literals(jaxpr, constvals)
    return jaxpr, constvals
```

```python
def _inline_literals(jaxpr: Jaxpr, consts: List[Any]) -> Tuple[Jaxpr,
List[Any]]:
  const_binders, other_binders = split_list(jaxpr.in_binders, len(consts))
  scalars = [type(x) in jax_types and not get_aval(x).shape for x in consts]
  new_const_binders, lit_binders = partition_list(scalars, const_binders)
  new_consts, lit_vals = partition_list(scalars, consts)
  literals = dict(zip(lit_binders, map(Lit, lit_vals)))
  new_eqns = [JaxprEqn(eqn.primitive, [literals.get(x, x) for x in
eqn.inputs],
                       eqn.params, eqn.out_binders) for eqn in jaxpr.eqns]
  new_outs = [literals.get(x, x) for x in jaxpr.outs]
  new_jaxpr = Jaxpr(new_const_binders + other_binders, new_eqns, new_outs)
  typecheck_jaxpr(new_jaxpr)
  return new_jaxpr, new_consts
```

The rules we need for `JaxprTrace.process_primitive` are essentially typing rules for primitive applications: given the primitive, its parameters, and types for the inputs, the rule must produce a type for the output, which is then packaged with the output `JaxprTracer`. We can use abstract evaluation rules for this same purpose, even though they can be more general (since abstract evaluation rules must accept ConcreteArray inputs, and since they need only return an upper bound on the set of possible outputs, they can produce ConcreteArray outputs as well). We'll reuse these abstract evaluation rules for the other jaxpr-producing trace machinery, where the potential extra generality is useful.

```python
def binop_abstract_eval(x: ShapedArray, y: ShapedArray) -> List[ShapedArray]:
  if not isinstance(x, ShapedArray) or not isinstance(y, ShapedArray):
    raise TypeError
  if raise_to_shaped(x) != raise_to_shaped(y): raise TypeError
  return [ShapedArray(x.shape, x.dtype)]

abstract_eval_rules[add_p] = binop_abstract_eval
abstract_eval_rules[mul_p] = binop_abstract_eval

def compare_abstract_eval(x: ShapedArray, y: ShapedArray) ->
List[ShapedArray]:
  if not isinstance(x, ShapedArray) or not isinstance(y, ShapedArray):
    raise TypeError
  if x.shape != y.shape: raise TypeError
  return [ShapedArray(x.shape, np.dtype('bool'))]
abstract_eval_rules[greater_p] = compare_abstract_eval
abstract_eval_rules[less_p] = compare_abstract_eval

def vectorized_unop_abstract_eval(x: ShapedArray) -> List[ShapedArray]:
  return [ShapedArray(x.shape, x.dtype)]

abstract_eval_rules[sin_p] = vectorized_unop_abstract_eval
abstract_eval_rules[cos_p] = vectorized_unop_abstract_eval
abstract_eval_rules[neg_p] = vectorized_unop_abstract_eval

def reduce_sum_abstract_eval(x: ShapedArray, *, axis: Tuple[int, ...]
                             ) -> List[ShapedArray]:
  axis_ = set(axis)
  new_shape = [d for i, d in enumerate(x.shape) if i not in axis_]
  return [ShapedArray(tuple(new_shape), x.dtype)]
abstract_eval_rules[reduce_sum_p] = reduce_sum_abstract_eval

def broadcast_abstract_eval(x: ShapedArray, *, shape: Sequence[int],
                            axes: Sequence[int]) -> List[ShapedArray]:
  return [ShapedArray(tuple(shape), x.dtype)]
abstract_eval_rules[broadcast_p] = broadcast_abstract_eval
```

To check our implementation of jaxprs, we can add a `make_jaxpr` transformation and a pretty-printer:

```python
from functools import lru_cache

@lru_cache()  # ShapedArrays are hashable
def make_jaxpr_v1(f, *avals_in):
  avals_in, in_tree = tree_flatten(avals_in)
  f, out_tree = flatten_fun(f, in_tree)

  builder = JaxprBuilder()
  with new_main(JaxprTrace, builder) as main:
    trace = JaxprTrace(main)
    tracers_in = [trace.new_arg(aval) for aval in avals_in]
    outs = f(*tracers_in)
    tracers_out = [full_raise(trace, out) for out in outs]
    jaxpr, consts = builder.build(tracers_in, tracers_out)
  return jaxpr, consts, out_tree()
```

▶ Show code cell source

```
jaxpr, consts, _ = make_jaxpr_v1(lambda x: 2. * x,
raise_to_shaped(get_aval(3.)))
print(jaxpr)
print(typecheck_jaxpr(jaxpr))
```

```
{ lambda a:float64[] .
  let b:float64[] = mul 2.0 a
  in ( b ) }
(float64[]) -> (float64[])
```

But there's a limitation here: because of how `find_top_trace` operates by data dependence, `make_jaxpr_v1` can't stage out all the primitive operations performed by the Python callable it's given. For example:

```
jaxpr, consts, _ = make_jaxpr_v1(lambda: mul(2., 2.))
print(jaxpr)
```

```
{ lambda  .
  let
  in ( 4.0 ) }
```

This is precisely the issue that [omnistaging](https://github.com/google/jax/pull/3370) fixed. We want to ensure that the `JaxprTrace` started by `make_jaxpr` is always applied, regardless of whether any inputs to `bind` are boxed in corresponding `JaxprTracer` instances. We can achieve this by employing the `dynamic_trace` global defined in Part 1:

```python
@contextmanager
def new_dynamic(main: MainTrace):
  global dynamic_trace
  prev_dynamic_trace, dynamic_trace = dynamic_trace, main
  try:
    yield
  finally:
    dynamic_trace = prev_dynamic_trace

@lru_cache()
def make_jaxpr(f: Callable, *avals_in: ShapedArray,
               ) -> Tuple[Jaxpr, List[Any], PyTreeDef]:
  avals_in, in_tree = tree_flatten(avals_in)
  f, out_tree = flatten_fun(f, in_tree)

  builder = JaxprBuilder()
  with new_main(JaxprTrace, builder) as main:
    with new_dynamic(main):
      trace = JaxprTrace(main)
      tracers_in = [trace.new_arg(aval) for aval in avals_in]
      outs = f(*tracers_in)
      tracers_out = [full_raise(trace, out) for out in outs]
      jaxpr, consts = builder.build(tracers_in, tracers_out)
  return jaxpr, consts, out_tree()

jaxpr, consts, _ = make_jaxpr(lambda: mul(2., 2.))
print(jaxpr)
```

```
{ lambda  .
  let a:float64[] = mul 2.0 2.0
  in ( a ) }
```

Using `dynamic_trace` this way is conceptually the same as stashing the current interpreter stack and starting a new one with the `JaxprTrace` at the bottom. That is, no interpreters lower in the stack than the `dynamic_trace` are applied (since `JaxprTrace.process_primitive` doesn't call `bind`), though if the Python callable being traced to a jaxpr itself uses transformations then those can be pushed onto the interpreter stack above the `JaxprTrace`. But temporarily stashing the interpreter stack would break up the system state. The `dynamic_trace` tag achieves the same goals while keeping the system state simpler.

That's it for jaxprs! With jaxprs in hand, we can implement the remaining major JAX features.

Part 3: `jit`, simplified

While `jit` has a transformation-like API in that it accepts a Python callable as an argument, under the hood it's really a higher-order primitive rather than a transformation. A primitive is *higher-order* when it's parameterized by a function.

On-the-fly ("final style") and staged ("initial style") processing

There are two options for how to handle higher-order primitives. Each requires a different approach to tracing and engenders different tradeoffs:

1. **On-the-fly processing, where `bind` takes a Python callable as an argument.** We defer forming a jaxpr until as late as possible, namely until we're running the final interpreter at the bottom of the interpreter stack. That way we can swap a `JaxprTrace` in at the bottom of the interpreter stack and thus stage out rather than execute all primitive operations. With this approach, transformations in the stack get applied as we execute the Python callable as usual. This approach can be very tricky to implement, but it's as general as possible because it allows higher-order primitives not to raise the abstraction level of their arguments and thus allows data-dependent Python control flow. We refer to this approach as using a "final-style higher-order primitive" employing the discharge-at-tracing-time "final-style transformations" we've used so far.
2. **Staged processing, where `bind` takes a jaxpr as an argument.** Before we call `bind`, in the primitive wrapper we can just use `make_jaxpr` to form a jaxpr up-front and be done with the Python callable entirely. In this case, `make_jaxpr` puts its `JaxprTrace` at the top of the interpreter stack, and no transformations lower in the stack, which might enter via closed-over Tracers, are applied to the Python callable as we trace it. (Transformations applied within the Python callable are applied as usual, being added to the stack above the JaxprTrace.) Instead, the transformations lower in the stack are later applied to the call primitive, and the call primitive's rules must then transform the jaxpr itself. Because we trace to a jaxpr up-front, this approach can't support data-dependent Python control flow, but it is more straightforward to implement. We refer to this kind of higher-order primitive as an "initial-style higher-order primitive", and say that its jaxpr-processing transformation rules are "initial-style transformation rules."

The latter approach fits for `jit` because we don't need to support data-dependent Python control flow in the user-provided Python callable, as the whole purpose of `jit` is to stage computation out of Python to be executed by XLA. (In contrast, `custom_jvp` is a higher-order primitive in which we want to support data-dependent Python control flow.)

Historically, we started using the "initial-style" and "final-style" terminology after reading the [typed tagless final interpreters] paper, and jokingly referring to JAX as an implementation of "untyped tagful final interpreters." We don't claim to carry over (or understand) any deep meaning behind these terms; we loosely use "initial style" to mean "build an AST and then transform it", and we use "final style" to mean "transform as we trace." But it's just imprecise yet sticky jargon.

With the initial-style approach, here's the user-facing `jit` wrapper:

```
def jit(f):
  def f_jitted(*args):
    avals_in = [raise_to_shaped(get_aval(x)) for x in args]
    jaxpr, consts, out_tree = make_jaxpr(f, *avals_in)
    outs = bind(xla_call_p, *consts, *args, jaxpr=jaxpr,
num_consts=len(consts))
    return tree_unflatten(out_tree, outs)
  return f_jitted

xla_call_p = Primitive('xla_call')
```

With any new primitive, we need to give it transformation rules, starting with its evaluation rule. When we evaluate an application of the `xla_call` primitive, we want to stage out out the computation to XLA. That involves translating the jaxpr to an XLA HLO program, transferring the argument values to the XLA device, executing the XLA program, and transferring back the results. We'll cache the XLA HLO compilation so that for each `jit`ted function it only needs to be performed once per argument shape and dtype signature.

First, some utilities.

```
class IDHashable:
  val: Any

  def __init__(self, val):
    self.val = val

  def __hash__(self) -> int:
    return id(self.val)

  def __eq__(self, other):
    return type(other) is IDHashable and id(self.val) == id(other.val)
```

Next, we'll define the evaluation rule for `xla_call`:

```python
from jax._src.lib import xla_bridge as xb
from jax._src.lib import xla_client as xc
xe = xc._xla
xops = xc._xla.ops

def xla_call_impl(*args, jaxpr: Jaxpr, num_consts: int):
  consts, args = args[:num_consts], args[num_consts:]
  hashable_consts = tuple(map(IDHashable, consts))
  execute = xla_callable(IDHashable(jaxpr), hashable_consts)
  return execute(*args)
impl_rules[xla_call_p] = xla_call_impl

@lru_cache()
def xla_callable(hashable_jaxpr: IDHashable,
                 hashable_consts: Tuple[IDHashable, ...]):
  jaxpr: Jaxpr = hashable_jaxpr.val
  typecheck_jaxpr(jaxpr)
  consts = [x.val for x in hashable_consts]
  in_avals = [v.aval for v in jaxpr.in_binders[len(consts):]]
  c = xc.XlaBuilder('xla_call')
  xla_consts = _xla_consts(c, consts)
  xla_params = _xla_params(c, in_avals)
  outs = jaxpr_subcomp(c, jaxpr, xla_consts + xla_params)
  out = xops.Tuple(c, outs)
  compiled = xb.get_backend(None).compile(
    xc._xla.mlir.xla_computation_to_mlir_module(c.build(out)))
  return partial(execute_compiled, compiled, [v.aval for v in jaxpr.outs])

def _xla_consts(c: xe.XlaBuilder, consts: List[Any]) -> List[xe.XlaOp]:
  unique_consts = {id(cnst): cnst for cnst in consts}
  xla_consts = {
      id_: xops.ConstantLiteral(c, cnst) for id_, cnst in
unique_consts.items()}
  return [xla_consts[id(cnst)] for cnst in consts]

def _xla_params(c: xe.XlaBuilder, avals_in: List[ShapedArray]) ->
List[xe.XlaOp]:
  return [xops.Parameter(c, i, _xla_shape(a)) for i, a in
enumerate(avals_in)]

def _xla_shape(aval: ShapedArray) -> xe.Shape:
  return xc.Shape.array_shape(xc.dtype_to_etype(aval.dtype), aval.shape)
```

The main action is in `xla_callable`, which compiles a jaxpr into an XLA HLO program using `jaxpr_subcomp`, then returns a callable which executes the compiled program:

```python
def jaxpr_subcomp(c: xe.XlaBuilder, jaxpr: Jaxpr, args: List[xe.XlaOp]
                  ) -> xe.XlaOp:
  env: Dict[Var, xe.XlaOp] = {}

  def read(x: Atom) -> xe.XlaOp:
    return env[x] if type(x) is Var else xops.Constant(c, np.asarray(x.val))

  def write(v: Var, val: xe.XlaOp) -> None:
    env[v] = val

  map(write, jaxpr.in_binders, args)
  for eqn in jaxpr.eqns:
    in_avals = [x.aval for x in eqn.inputs]
    in_vals = map(read, eqn.inputs)
    rule = xla_translations[eqn.primitive]
    out_vals = rule(c, in_avals, in_vals, **eqn.params)
    map(write, eqn.out_binders, out_vals)
  return map(read, jaxpr.outs)

def execute_compiled(compiled, out_avals, *args):
  input_bufs = [input_handlers[type(x)](x) for x in args]
  out_bufs = compiled.execute(input_bufs)
  return [handle_result(aval, buf) for aval, buf in zip(out_avals, out_bufs)]

default_input_handler = xb.get_backend(None).buffer_from_pyval
input_handlers = {ty: default_input_handler for ty in
                  [bool, int, float, np.ndarray, np.float64, np.float32]}

def handle_result(aval: ShapedArray, buf):
  del aval  # Unused for now
  return np.asarray(buf)

xla_translations = {}
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun
for more info.)
```

Notice that `jaxpr_subcomp` has the structure of a simple interpreter. That's a common pattern: the way we process jaxprs is usually with an interpreter. And as with any interpreter, we need an interpretation rule for each primitive:

```python
def direct_translation(op, c, in_avals, in_vals):
  del c, in_avals
  return [op(*in_vals)]

xla_translations[add_p] = partial(direct_translation, xops.Add)
xla_translations[mul_p] = partial(direct_translation, xops.Mul)
xla_translations[neg_p] = partial(direct_translation, xops.Neg)
xla_translations[sin_p] = partial(direct_translation, xops.Sin)
xla_translations[cos_p] = partial(direct_translation, xops.Cos)
xla_translations[greater_p] = partial(direct_translation, xops.Gt)
xla_translations[less_p] = partial(direct_translation, xops.Lt)

def reduce_sum_translation(c, in_avals, in_vals, *, axis):
  (x_aval,), (x,) = in_avals, in_vals
  zero = xops.ConstantLiteral(c, np.array(0, x_aval.dtype))
  subc = xc.XlaBuilder('add')
  shape = _xla_shape(ShapedArray((), x_aval.dtype))
  xops.Add(xops.Parameter(subc, 0, shape), xops.Parameter(subc, 1, shape))
  return [xops.Reduce(c, [x], [zero], subc.build(), axis)]
xla_translations[reduce_sum_p] = reduce_sum_translation

def broadcast_translation(c, in_avals, in_vals, *, shape, axes):
  x, = in_vals
  dims_complement = [i for i in range(len(shape)) if i not in axes]
  return [xops.BroadcastInDim(x, shape, dims_complement)]
xla_translations[broadcast_p] = broadcast_translation
```

With that, we can now use `jit` to stage out, compile, and execute programs with XLA!

```python
@jit
def f(x, y):
  print('tracing!')
  return sin(x) * cos(y)
```

```python
z = f(3., 4.)  # 'tracing!' prints the first time
print(z)
```

```
tracing!
-0.09224219304455371
```

```python
z = f(4., 5.)  # 'tracing!' doesn't print, compilation cache hit!
print(z)
```

```
-0.21467624978306993
```

```python
@jit
def f(x):
  return reduce_sum(x, axis=0)

print(f(np.array([1., 2., 3.])))
```

```
6.0
```

```python
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z

def deriv(f):
  return lambda x: jvp(f, (x,), (1.,))[1]

print(    deriv(deriv(f))(3.))
print(jit(deriv(deriv(f)))(3.))
```

```
0.2822400161197344
0.2822400161197344
```

Instead of implementing `jit` to first trace to a jaxpr and then to lower the jaxpr to XLA HLO, it might appear that we could have skipped the jaxpr step and just lowered to HLO while tracing. That is, perhaps we could have instead implemented `jit` with a `Trace` and `Tracer` that appended to the XLA HLO graph incrementally on each primitive bind. That's correct for now, but won't be possible when we introduce compiled SPMD computations because there we must know the number of replicas needed before compiling the program.

We haven't yet defined any transformation rules for `xla_call_p` other than its evaluation rule. That is, we can't yet do `vmap`-of-`jit` or `jvp`-of-`jit` or even `jit`-of-`jit`. Instead `jit` has to be at the "top level." Let's fix that!

```python
def xla_call_jvp_rule(primals, tangents, *, jaxpr, num_consts):
  del num_consts  # Unused
  new_jaxpr, new_consts = jvp_jaxpr(jaxpr)
  outs = bind(xla_call_p, *new_consts, *primals, *tangents, jaxpr=new_jaxpr,
              num_consts=len(new_consts))
  n = len(outs) // 2
  primals_out, tangents_out = outs[:n], outs[n:]
  return primals_out, tangents_out
jvp_rules[xla_call_p] = xla_call_jvp_rule

@lru_cache()
def jvp_jaxpr(jaxpr: Jaxpr) -> Tuple[Jaxpr, List[Any]]:
  def jvp_traceable(*primals_and_tangents):
    n = len(primals_and_tangents) // 2
    primals, tangents = primals_and_tangents[:n], primals_and_tangents[n:]
    return jvp(jaxpr_as_fun(jaxpr), primals, tangents)

  in_avals = [v.aval for v in jaxpr.in_binders]
  new_jaxpr, new_consts, _ = make_jaxpr(jvp_traceable, *in_avals, *in_avals)
  return new_jaxpr, new_consts
```

```python
def xla_call_vmap_rule(axis_size, vals_in, dims_in, *, jaxpr, num_consts):
  del num_consts  # Unused
  new_jaxpr, new_consts = vmap_jaxpr(jaxpr, axis_size, tuple(dims_in))
  outs = bind(xla_call_p, *new_consts, *vals_in, jaxpr=new_jaxpr,
              num_consts=len(new_consts))
  return outs, [0] * len(outs)
vmap_rules[xla_call_p] = xla_call_vmap_rule

@lru_cache()
def vmap_jaxpr(jaxpr: Jaxpr, axis_size: int, bdims_in: Tuple[BatchAxis, ...]
               ) -> Tuple[Jaxpr, List[Any]]:
  vmap_traceable = vmap(jaxpr_as_fun(jaxpr), tuple(bdims_in))
  in_avals = [unmapped_aval(axis_size, d, v.aval)
              for v, d in zip(jaxpr.in_binders, bdims_in)]
  new_jaxpr, new_consts, _ = make_jaxpr(vmap_traceable, *in_avals)
  return new_jaxpr, new_consts

def unmapped_aval(axis_size: int, batch_dim: BatchAxis, aval: ShapedArray
                  ) -> ShapedArray:
  if batch_dim is not_mapped:
    return aval
  else:
    shape = list(aval.shape)
    shape.insert(batch_dim, axis_size)
    return ShapedArray(tuple(shape), aval.dtype)
```

```python
def xla_call_abstract_eval_rule(*in_types, jaxpr, num_consts):
  del num_consts  # Unused
  jaxpr_type = typecheck_jaxpr(jaxpr)
  if not all(t1 == t2 for t1, t2 in zip(jaxpr_type.in_types, in_types)):
    raise TypeError
  return jaxpr_type.out_types
abstract_eval_rules[xla_call_p] = xla_call_abstract_eval_rule

def xla_call_translation(c, in_avals, in_vals, *, jaxpr, num_consts):
  del num_consts  # Only used at top-level.
  # Calling jaxpr_subcomp directly would inline. We generate a Call HLO
instead.
  subc = xc.XlaBuilder('inner xla_call')
  xla_params = _xla_params(subc, in_avals)
  outs = jaxpr_subcomp(subc, jaxpr, xla_params)
  subc = subc.build(xops.Tuple(subc, outs))
  return destructure_tuple(c, xops.Call(c, subc, in_vals))
xla_translations[xla_call_p] = xla_call_translation

def destructure_tuple(c, tup):
  num_elements = len(c.get_shape(tup).tuple_shapes())
  return [xops.GetTupleElement(tup, i) for i in range(num_elements)]
```

```python
@jit
def f(x):
  print('tracing!')
  y = sin(x) * 2.
  z = - y + x
  return z

x, xdot = 3., 1.
y, ydot = jvp(f, (x,), (xdot,))
print(y)
print(ydot)
```

```
tracing!
2.7177599838802657
2.979984993200891
```

```python
y, ydot = jvp(f, (x,), (xdot,))  # 'tracing!' not printed
```

```python
ys = vmap(f, (0,))(np.arange(3.))
print(ys)
```

```
[ 0.         -0.68294197  0.18140515]
```

One piece missing is device memory persistence for arrays. That is, we've defined `handle_result` to transfer results back to CPU memory as NumPy arrays, but it's often preferable to avoid transferring results just to transfer them back for the next operation. We can do that by introducing a `DeviceArray` class, which can wrap XLA buffers and otherwise duck-type `numpy.ndarray`s:

```python
def handle_result(aval: ShapedArray, buf):  # noqa: F811
  return DeviceArray(aval, buf)

class DeviceArray:
  buf: Any
  aval: ShapedArray

  def __init__(self, aval, buf):
    self.aval = aval
    self.buf = buf

  dtype = property(lambda self: self.aval.dtype)
  shape = property(lambda self: self.aval.shape)
  ndim  = property(lambda self: self.aval.ndim)

  def __array__(self): return np.asarray(self.buf)
  def __repr__(self):  return repr(np.asarray(self.buf))
  def __str__(self):   return str(np.asarray(self.buf))

  _neg = staticmethod(neg)
  _add = staticmethod(add)
  _radd = staticmethod(add)
  _mul = staticmethod(mul)
  _rmul = staticmethod(mul)
  _gt = staticmethod(greater)
  _lt = staticmethod(less)
input_handlers[DeviceArray] = lambda x: x.buf

jax_types.add(DeviceArray)
```

```python
@jit
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z

x, xdot = 3., 1.
y, ydot = jvp(f, (x,), (xdot,))
print(y)
print(ydot)
```

```
2.7177599838802657
2.979984993200891
```

▶ Show code cell source

Part 4: `linearize` and `vjp` (and `grad`!)

The `linearize` and `vjp` autodiff functions are built on `jvp`, but involve jaxprs as well. That's because both involve staging out, or delaying, computation.

`linearize`

In the case of `linearize`, we want to stage out the linear part of a `jvp` computation. That is, in terms of [Haskell-like type signatures](), if we have `jvp : (a -> b) -> (a, T a) -> (b, T b)`, then we write `linearize : (a -> b) -> a -> (b, T a -o T b)`, using `T a` to mean "the tangent type of `a`" and using the "lollipop" `-o` rather than the arrow `->` to indicate a *linear* function. We define the semantics of `linearize` in terms of `jvp` too:

```
y, f_lin = linearize(f, x)
y_dot = f_lin(x_dot)
```

gives the same result for `(y, y_dot)` as

```
y, y_dot = jvp(f, (x,), (x_dot,))
```

where the application of `f_lin` does not redo any of the linearization work. We'll represent the delayed linear part `f_lin : T a -o T b` as a jaxpr.

Tangentially, now that we have linear arrows `-o`, we can provide a slightly more informative type for `jvp`:

```
jvp : (a -> b) -> (UnrestrictedUse a, T a) -o (UnrestrictedUse b, T b)
```

Here we're writing `UnrestrictedUse` just to indicate that we have a special pair where the first element can be used in an unrestricted (nonlinear) way. In conjunction with the linear arrow, this notation is just meant to express that the function `jvp f` uses its first input in a nonlinear way but its second input in a linear way, producing a corresponding nonlinear output (which can be used in a nonlinear way) paired with a linear output. This more refined type signature encodes the data dependencies in `jvp f`, which are useful for partial evaluation.

To build the `f_lin` jaxpr from a JVP, we need to perform partial evaluation: we evaluate all the primal values as we trace, but stage the tangent computations into a jaxpr. This is our second way to build jaxprs. But where `make_jaxpr` and its underlying `JaxprTrace`/`JaxprTracer` interpreters aim to stage out every primitive bind, this second approach stages out only those primitive binds with a data dependence on tangent inputs.

First, some utilities:

```python
def split_half(lst: List[Any]) -> Tuple[List[Any], List[Any]]:
  assert not len(lst) % 2
  return split_list(lst, len(lst) // 2)

def merge_lists(which: List[bool], l1: List[Any], l2: List[Any]) ->
List[Any]:
  l1, l2 = iter(l1), iter(l2)
  out = [next(l2) if b else next(l1) for b in which]
  assert next(l1, None) is next(l2, None) is None
  return out
```

Next, we'll write `linearize` by combining `jvp` together with a general partial evaluation transformation, to be added next:

```python
def linearize_flat(f, *primals_in):
  pvals_in = ([PartialVal.known(x) for x in primals_in] +
              [PartialVal.unknown(vspace(get_aval(x))) for x in primals_in])
  def f_jvp(*primals_tangents_in):
    primals_out, tangents_out = jvp(f, *split_half(primals_tangents_in))
    return [*primals_out, *tangents_out]
  jaxpr, pvals_out, consts = partial_eval_flat(f_jvp, pvals_in)
  primal_pvals, _ = split_half(pvals_out)
  assert all(pval.is_known for pval in primal_pvals)
  primals_out = [pval.const for pval in primal_pvals]
  f_lin = lambda *tangents: eval_jaxpr(jaxpr, [*consts, *tangents])
  return primals_out, f_lin

def linearize(f, *primals_in):
  primals_in_flat, in_tree = tree_flatten(primals_in)
  f, out_tree = flatten_fun(f, in_tree)
  primals_out_flat, f_lin_flat = linearize_flat(f, *primals_in_flat)
  primals_out = tree_unflatten(out_tree(), primals_out_flat)

  def f_lin(*tangents_in):
    tangents_in_flat, in_tree2 = tree_flatten(tangents_in)
    if in_tree != in_tree2: raise TypeError
    tangents_out_flat = f_lin_flat(*tangents_in_flat)
    return tree_unflatten(out_tree(), tangents_out_flat)

  return primals_out, f_lin

def vspace(aval: ShapedArray) -> ShapedArray:
  return raise_to_shaped(aval)  # TODO handle integers?
```

Now we turn to the general partial evaluation transformation. The goal is to accept a Python callable and a list of inputs, some known and some unknown, and to produce (1) all the outputs which can be computed from the known inputs, together with (2) a jaxpr representing the part of the Python callable's computation which can only be performed after the remaining inputs are known.

This transformation is tricky to summarize in a type signature. If we assume the input function's type signature is `(a1, a2) -> (b1, b2)`, where `a1` and `a2` represent the known and unknown inputs, respectively, and where `b1` only has a data dependency on `a1` while `b2` has some data dependency on `a2`, then we might write

```
partial_eval : ((a1, a2) -> (b1, b2)) -> a1 -> exists r. (b1, r, (r, a2) ->
b2)
```

In words, given values for the inputs of type `a1`, `partial_eval` produces the outputs of type `b1` along with “residual” values of existentially-quantified type `r` representing the intermediates required to complete the computation in the second stage. It also produces a function of type `(r, a2) -> b2` which accepts the residual values as well as the remaining inputs and produces the remaining outputs.

We like to think of partial evaluation as “unzipping” one computation into two. For example, consider this jaxpr:

```
{ lambda a:float64[] .
  let b:float64[] = sin a
      c:float64[] = neg b
  in ( c ) }
```

A jaxpr for the JVP would look like:

```
{ lambda a:float64[] b:float64[] .
  let c:float64[] = sin a
      d:float64[] = cos a
      e:float64[] = mul d b
      f:float64[] = neg c
      g:float64[] = neg e
  in ( f, g ) }
```

If we imagine applying partial evaluation to this jaxpr with the first input known and the second unknown, we end up ‘unzipping’ the JVP jaxpr into primal and tangent jaxprs:

```
{ lambda a:float64[] .
  let c:float64[] = sin a
      d:float64[] = cos a
      f:float64[] = neg c
  in ( f, d ) }
```

```
{ lambda d:float64[] b:float64[] .
  let e:float64[] = mul d b
      g:float64[] = neg e
  in ( g ) }
```

This second jaxpr represents the linear computation that we want from `linearize`.

However, unlike in this jaxpr example, we want the computation on known values to occur while evaluating the input Python callable. That is, rather than forming a jaxpr for the entire function `(a1, a2) -> (b1, b2)`, staging all operations out of Python first before sorting out what can be evaluated now and what must be delayed, we want only to form a jaxpr for those operations that *must* be delayed due to a dependence on unknown inputs. In the context of automatic differentiation, this is the feature that ultimately enables us to handle functions like `grad(lambda x: x**2 if x > 0 else 0.)`. Python control flow works because partial evaluation keeps the primal computation in Python. As a consequence, our `Trace` and `Tracer` subclasses must on the fly sort out what can be evaluated and what must be staged out into a jaxpr.

First, we start with a `PartialVal` class, which represents a value that can be either known or unknown:

```
class PartialVal(NamedTuple):
  aval: ShapedArray
  const: Optional[Any]

  @classmethod
  def known(cls, val: Any):
    return PartialVal(get_aval(val), val)

  @classmethod
  def unknown(cls, aval: ShapedArray):
    return PartialVal(aval, None)

  is_known   = property(lambda self: self.const is not None)
  is_unknown = property(lambda self: self.const is     None)
```

Partial evaluation will take a list of `PartialVal`s representing inputs, and return a list of `PartialVal` outputs along with a jaxpr representing the delayed computation:

```
def partial_eval_flat(f: Callable, pvals_in: List[PartialVal]
                      ) -> Tuple[Jaxpr, List[PartialVal], List[Any]]:
  with new_main(PartialEvalTrace) as main:
    trace = PartialEvalTrace(main)
    tracers_in = [trace.new_arg(pval) for pval in pvals_in]
    outs = f(*tracers_in)
    tracers_out = [full_raise(trace, out) for out in outs]
    pvals_out = [t.pval for t in tracers_out]
    unk_tracers_in  = [t for t in tracers_in  if t.pval.is_unknown]
    unk_tracers_out = [t for t in tracers_out if t.pval.is_unknown]
    jaxpr, consts = tracers_to_jaxpr(unk_tracers_in, unk_tracers_out)
  return jaxpr, pvals_out, consts
```

Next we need to implement `PartialEvalTrace` and its `PartialEvalTracer`. This interpreter will build a jaxpr on the fly while tracking data dependencies. To do so, it builds a bipartite directed acyclic graph (DAG) between `PartialEvalTracer` nodes, representing staged-out values, and `JaxprRecipe` nodes, representing formulas for how to compute some values from others. One kind of recipe is a `JaxprEqnRecipe`, corresponding to a `JaxprEqn`'s primitive application, but we also have recipe types for constants and lambda binders:

```python
from weakref import ref, ReferenceType

class LambdaBindingRecipe(NamedTuple):
  pass

class ConstRecipe(NamedTuple):
  val: Any

class JaxprEqnRecipe(NamedTuple):
  prim: Primitive
  tracers_in: List['PartialEvalTracer']
  params: Dict[str, Any]
  avals_out: List[ShapedArray]
  tracer_refs_out: List['ReferenceType[PartialEvalTracer]']

JaxprRecipe = Union[LambdaBindingRecipe, ConstRecipe, JaxprEqnRecipe]
```

```python
class PartialEvalTracer(Tracer):
  pval: PartialVal
  recipe: Optional[JaxprRecipe]

  def __init__(self, trace, pval, recipe):
    self._trace = trace
    self.pval = pval
    self.recipe = recipe

  aval = property(lambda self: self.pval.aval)

  def full_lower(self):
    if self.pval.is_known:
      return full_lower(self.pval.const)
    return self
```

The `PartialEvalTrace` contains the logic for constructing the graph of `JaxprRecipe`s and `PartialEvalTracer`s. Each argument corresponds to a `LambdaBindingRecipe` leaf node, and each constant is a `ConstRecipe` leaf node holding a reference to the constant. All other tracers and recipes come from `process_primitive`, which forms tracers with `JaxprEqnRecipe`s.

For most primitives, the `process_primitive` logic is straightforward: if all inputs are known then we can bind the primitive on the known values (evaluating it in Python) and avoid forming tracers corresponding to the output. If instead any input is unknown then we instead stage out into a `JaxprEqnRecipe` representing the primitive application. To build the tracers representing unknown outputs, we need avals, which we get from the abstract eval rules. (Notice that tracers reference `JaxprEqnRecipe`s, and `JaxprEqnRecipe`s reference tracers; we avoid circular garbage by using weakrefs.)

That `process_primitive` logic applies to most primitives, but `xla_call_p` requires recursive treatment. So we special-case its rule in a `partial_eval_rules` dict.

```python
class PartialEvalTrace(Trace):
  def new_arg(self, pval: PartialVal) -> Any:
    return PartialEvalTracer(self, pval, LambdaBindingRecipe())

  def lift(self, val: Any) -> PartialEvalTracer:
    return PartialEvalTracer(self, PartialVal.known(val), None)
  pure = lift

  def instantiate_const(self, tracer: PartialEvalTracer) ->
PartialEvalTracer:
    if tracer.pval.is_unknown:
      return tracer
    else:
      pval = PartialVal.unknown(raise_to_shaped(tracer.aval))
      return PartialEvalTracer(self, pval, ConstRecipe(tracer.pval.const))

  def process_primitive(self, primitive, tracers, params):
    if all(t.pval.is_known for t in tracers):
      return bind(primitive, *map(full_lower, tracers), **params)
    rule = partial_eval_rules.get(primitive)
    if rule: return rule(self, tracers, **params)
    tracers_in = [self.instantiate_const(t) for t in tracers]
    avals_in = [t.aval for t in tracers_in]
    avals_out = abstract_eval_rules[primitive](*avals_in, **params)
    tracers_out = [PartialEvalTracer(self, PartialVal.unknown(aval), None)
                   for aval in avals_out]
    eqn = JaxprEqnRecipe(primitive, tracers_in, params, avals_out,
                         map(ref, tracers_out))
    for t in tracers_out: t.recipe = eqn
    return tracers_out

partial_eval_rules = {}
```

Now that we can build graph representations of jaxprs with `PartialEvalTrace`, we need a mechanism to convert the graph representation to a standard jaxpr. The jaxpr corresponds to a topological sort of the graph.

```python
def tracers_to_jaxpr(tracers_in: List[PartialEvalTracer],
                     tracers_out: List[PartialEvalTracer]):
  tracer_to_var: Dict[int, Var] = {id(t): Var(raise_to_shaped(t.aval))
                                   for t in tracers_in}
  constvar_to_val: Dict[int, Any] = {}
  constid_to_var: Dict[int, Var] = {}
  processed_eqns: Set[int] = set()
  eqns: List[JaxprEqn] = []
  for t in toposort(tracers_out, tracer_parents):
    if isinstance(t.recipe, LambdaBindingRecipe):
      assert id(t) in set(map(id, tracers_in))
    elif isinstance(t.recipe, ConstRecipe):
      val = t.recipe.val
      var = constid_to_var.get(id(val))
      if var is None:
        aval = raise_to_shaped(get_aval(val))
        var = constid_to_var[id(val)] = Var(aval)
        constvar_to_val[var] = val
      tracer_to_var[id(t)] = var
    elif isinstance(t.recipe, JaxprEqnRecipe):
      if id(t.recipe) not in processed_eqns:
        eqns.append(recipe_to_eqn(tracer_to_var, t.recipe))
        processed_eqns.add(id(t.recipe))
    else:
      raise TypeError(t.recipe)

  constvars, constvals = unzip2(constvar_to_val.items())
  in_binders = constvars + [tracer_to_var[id(t)] for t in tracers_in]
  out_vars = [tracer_to_var[id(t)] for t in tracers_out]
  jaxpr = Jaxpr(in_binders, eqns, out_vars)
  typecheck_jaxpr(jaxpr)
  return jaxpr, constvals

def recipe_to_eqn(tracer_to_var: Dict[int, Var], recipe: JaxprEqnRecipe
                  ) -> JaxprEqn:
  inputs = [tracer_to_var[id(t)] for t in recipe.tracers_in]
  out_binders = [Var(aval) for aval in recipe.avals_out]
  for t_ref, var in zip(recipe.tracer_refs_out, out_binders):
    if t_ref() is not None: tracer_to_var[id(t_ref())] = var
  return JaxprEqn(recipe.prim, inputs, recipe.params, out_binders)

def tracer_parents(t: PartialEvalTracer) -> List[PartialEvalTracer]:
  return t.recipe.tracers_in if isinstance(t.recipe, JaxprEqnRecipe) else []
```

▶ Show code cell source

Now we can linearize!

```python
y, sin_lin = linearize(sin, 3.)
print(y, sin(3.))
print(sin_lin(1.), cos(3.))
```

```
0.1411200080598672 0.1411200080598672
-0.9899924966004454 -0.9899924966004454
```

To handle `linearize`-of-`jit`, we still need to write a partial evaluation rule for `xla_call_p`. Other than tracer bookkeeping, the main task is to perform partial evaluation of a jaxpr, 'unzipping' it into two jaxprs.

There are actually two rules to write: one for trace-time partial evaluation, which we'll call `xla_call_partial_eval`, and one for partial evaluation of jaxprs, which we'll call `xla_call_peval_eqn`.

```
def xla_call_partial_eval(trace, tracers, *, jaxpr, num_consts):
  del num_consts  # Unused
  in_unknowns = [not t.pval.is_known for t in tracers]
  jaxpr1, jaxpr2, out_unknowns, num_res = partial_eval_jaxpr(jaxpr,
in_unknowns)
  known_tracers, unknown_tracers = partition_list(in_unknowns, tracers)
  known_vals = [t.pval.const for t in known_tracers]
  outs1_res = bind(xla_call_p, *known_vals, jaxpr=jaxpr1, num_consts=0)
  outs1, res = split_list(outs1_res, len(jaxpr1.outs) - num_res)
  res_tracers = [trace.instantiate_const(full_raise(trace, x)) for x in res]
  outs2 = [PartialEvalTracer(trace, PartialVal.unknown(v.aval), None)
           for v in jaxpr2.outs]
  eqn = JaxprEqnRecipe(xla_call_p, res_tracers + unknown_tracers,
                       dict(jaxpr=jaxpr2, num_consts=0),
                       [v.aval for v in jaxpr2.outs], map(ref, outs2))
  for t in outs2: t.recipe = eqn
  return merge_lists(out_unknowns, outs1, outs2)
partial_eval_rules[xla_call_p] = xla_call_partial_eval

def partial_eval_jaxpr(jaxpr: Jaxpr, in_unknowns: List[bool],
                       instantiate: Optional[List[bool]] = None,
                       ) -> Tuple[Jaxpr, Jaxpr, List[bool], int]:
  env: Dict[Var, bool] = {}
  residuals: Set[Var] = set()

  def read(x: Atom) -> bool:
    return type(x) is Var and env[x]

  def write(unk: bool, v: Var) -> None:
    env[v] = unk

  def new_res(x: Atom) -> Atom:
    if type(x) is Var: residuals.add(x)
    return x

  eqns1, eqns2 = [], []
  map(write, in_unknowns, jaxpr.in_binders)
  for eqn in jaxpr.eqns:
    unks_in = map(read, eqn.inputs)
    rule = partial_eval_jaxpr_rules.get(eqn.primitive)
    if rule:
      eqn1, eqn2, unks_out, res = rule(unks_in, eqn)
      eqns1.append(eqn1); eqns2.append(eqn2); residuals.update(res)
      map(write, unks_out, eqn.out_binders)
    elif any(unks_in):
      inputs = [v if unk else new_res(v) for unk, v in zip(unks_in,
eqn.inputs)]
      eqns2.append(JaxprEqn(eqn.primitive, inputs, eqn.params,
eqn.out_binders))
      map(partial(write, True), eqn.out_binders)
    else:
      eqns1.append(eqn)
      map(partial(write, False), eqn.out_binders)
  out_unknowns = map(read, jaxpr.outs)
  if instantiate is not None:
    for v, uk, inst in zip(jaxpr.outs, out_unknowns, instantiate):
      if inst and not uk: new_res(v)
    out_unknowns = map(op.or_, out_unknowns, instantiate)

  residuals, num_res = list(residuals), len(residuals)
  assert all(type(v) is Var for v in residuals), residuals
```

```
  ins1, ins2 = partition_list(in_unknowns, jaxpr.in_binders)
  outs1, outs2 = partition_list(out_unknowns, jaxpr.outs)

  jaxpr1 = Jaxpr(ins1, eqns1, outs1 + residuals)
  jaxpr2 = Jaxpr(residuals + ins2, eqns2, outs2)
  typecheck_partial_eval_jaxpr(jaxpr, in_unknowns, out_unknowns, jaxpr1,
jaxpr2)

  return jaxpr1, jaxpr2, out_unknowns, num_res

def typecheck_partial_eval_jaxpr(jaxpr, unks_in, unks_out, jaxpr1, jaxpr2):
  jaxprty = typecheck_jaxpr(jaxpr)    # (a1,  a2) -> (b1, b2 )
  jaxpr1ty = typecheck_jaxpr(jaxpr1)  #  a1       -> (b1, res)
  jaxpr2ty = typecheck_jaxpr(jaxpr2)  # (res, a2) -> b2

  a1, a2 = partition_list(unks_in, jaxprty.in_types)
  b1, b2 = partition_list(unks_out, jaxprty.out_types)
  b1_, res = split_list(jaxpr1ty.out_types, len(b1))
  res_, a2_ = split_list(jaxpr2ty.in_types, len(res))
  b2_ = jaxpr2ty.out_types

  if jaxpr1ty.in_types != a1: raise TypeError
  if jaxpr2ty.out_types != b2: raise TypeError
  if b1 != b1_: raise TypeError
  if res != res_: raise TypeError
  if a2 != a2_: raise TypeError
  if b2 != b2_: raise TypeError

partial_eval_jaxpr_rules = {}

def xla_call_peval_eqn(unks_in: List[bool], eqn: JaxprEqn,
                       ) -> Tuple[JaxprEqn, JaxprEqn, List[bool], List[Var]]:
  jaxpr = eqn.params['jaxpr']
  jaxpr1, jaxpr2, unks_out, num_res = partial_eval_jaxpr(jaxpr, unks_in)
  ins1, ins2 = partition_list(unks_in, eqn.inputs)
  out_binders1, out_binders2 = partition_list(unks_out, eqn.out_binders)
  residuals = [Var(v.aval) for v in jaxpr2.in_binders[:num_res]]
  eqn1 = JaxprEqn(xla_call_p, ins1, dict(jaxpr=jaxpr1, num_consts=0),
                  out_binders1 + residuals)
  eqn2 = JaxprEqn(xla_call_p, residuals + ins2,
                  dict(jaxpr=jaxpr2, num_consts=0), out_binders2)
  return eqn1, eqn2, unks_out, residuals
partial_eval_jaxpr_rules[xla_call_p] = xla_call_peval_eqn
```

With that, we can compose `linearize` and `jit` however we like:

```
@jit
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z

y, f_lin = linearize(f, 3.)
y_dot = f_lin(1.)
print(y, y_dot)
```

```
2.7177599838802657 2.979984993200891
```

```python
@jit
def f(x):
  y = sin(x) * 2.
  z = g(x, y)
  return z

@jit
def g(x, y):
  return cos(x) + y

y, f_lin = linearize(f, 3.)
y_dot = f_lin(1.)
print(y, y_dot)
```

```
-0.7077524804807109 -2.121105001260758
```

`vjp` and `grad`

The `vjp` transformation works a lot like linearize. Its type signature is analogous:

```
linearize : (a -> b) -> a -> (b, T a -o T b)
vjp       : (a -> b) -> a -> (b, T b -o T a)
```

The only difference is that we transpose the linear part of the computation before returning it, so that it goes from type `T a -o T b` to type `T b -o T a`. That is, we'll implement `vjp` as, essentially,

```python
def vjp(f, x):
  y, f_lin = linearize(f, x)
  f_vjp = lambda y_bar: transpose(f_lin)(y_bar)
  return y, f_vjp
```

Since we have the linear computation as a jaxpr, not just a Python callable, we can implement the transpose transformation as a jaxpr interpreter.

```python
def vjp_flat(f, *primals_in):
  pvals_in = ([PartialVal.known(x) for x in primals_in] +
              [PartialVal.unknown(vspace(get_aval(x))) for x in primals_in])
  primal_pvals_in, tangent_pvals_in = split_half(pvals_in)
  def f_jvp(*primals_tangents_in):
    primals_out, tangents_out = jvp(f, *split_half(primals_tangents_in))
    return [*primals_out, *tangents_out]
  jaxpr, pvals_out, consts = partial_eval_flat(f_jvp, pvals_in)  # linearize
  primal_pvals, _ = split_half(pvals_out)
  assert all(pval.is_known for pval in primal_pvals)
  primals_out = [pval.const for pval in primal_pvals]
  transpose_inputs = consts + [UndefPrimal(p.aval) for p in tangent_pvals_in]
  f_vjp = lambda *cts: eval_jaxpr_transposed(jaxpr, transpose_inputs, cts)
  return primals_out, f_vjp

def vjp(f, *primals_in):
  primals_in_flat, in_tree = tree_flatten(primals_in)
  f, out_tree = flatten_fun(f, in_tree)
  primals_out_flat, f_vjp_flat = vjp_flat(f, *primals_in_flat)
  primals_out = tree_unflatten(out_tree(), primals_out_flat)

  def f_vjp(*cotangents_out):
    cotangents_out_flat, _ = tree_flatten(cotangents_out)
    cotangents_in_flat = f_vjp_flat(*cotangents_out_flat)
    return tree_unflatten(in_tree, cotangents_in_flat)

  return primals_out, f_vjp

class UndefPrimal(NamedTuple):
  aval: ShapedArray

register_pytree_node(UndefPrimal,
                     lambda u: (u.aval, ()),
                     lambda aval, _: UndefPrimal(aval))
```

We use `UndefPrimal` instances to indicate which arguments with respect to which we want to transpose. These arise because in general, being explicit about closed-over values, we want to transpose functions of type `a -> b -o c` to functions of type `a -> c -o b`. Even more generally, the inputs with respect to which the function is linear could be scattered through the argument list. So we indicate the linear positions using `UndefPrimal`. We register `UndefPrimal` as a pytree node because the pytree mechanism gives a handy way to prune these placeholders out of argument lists.

Next, we can write `eval_jaxpr_transposed`, along with transpose rules for all primitives which can be linear in at least one argument:

```python
# NB: the analogous function in JAX is called 'backward_pass'
def eval_jaxpr_transposed(jaxpr: Jaxpr, args: List[Any], cotangents:
List[Any]
                          ) -> List[Any]:
  primal_env: Dict[Var, Any] = {}
  ct_env: Dict[Var, Any] = {}

  def read_primal(x: Atom) -> Any:
    return primal_env.get(x, UndefPrimal(x.aval)) if type(x) is Var else
x.val

  def write_primal(v: Var, val: Any) -> None:
    if type(val) is not UndefPrimal:
      primal_env[v] = val

  def read_cotangent(v: Var) -> Any:
    return ct_env.pop(v, np.zeros(v.aval.shape, v.aval.dtype))

  def write_cotangent(x: Atom, val: Any):
    if type(x) is Var and val is not None:
      ct_env[x] = add(ct_env[x], val) if x in ct_env else val

  map(write_primal, jaxpr.in_binders, args)
  map(write_cotangent, jaxpr.outs, cotangents)
  for eqn in jaxpr.eqns[::-1]:
    primals_in = map(read_primal, eqn.inputs)
    cts_in = map(read_cotangent, eqn.out_binders)
    rule = transpose_rules[eqn.primitive]
    cts_out = rule(cts_in, *primals_in, **eqn.params)
    map(write_cotangent, eqn.inputs, cts_out)

  return [read_cotangent(v) for v, x in zip(jaxpr.in_binders, args)
          if type(x) is UndefPrimal]

transpose_rules = {}
```

```
def mul_transpose_rule(cts, x, y):
  z_bar, = cts
  assert (type(x) is UndefPrimal) ^ (type(y) is UndefPrimal)
  return [mul(z_bar, y), None] if type(x) is UndefPrimal else [None, mul(x,
z_bar)]
transpose_rules[mul_p] = mul_transpose_rule

def neg_transpose_rule(cts, x):
  ybar, = cts
  assert type(x) is UndefPrimal
  return [neg(ybar)]
transpose_rules[neg_p] = neg_transpose_rule

def add_transpose_rule(cts, x, y):
  z_bar, = cts
  return [z_bar, z_bar]
transpose_rules[add_p] = add_transpose_rule

def reduce_sum_transpose_rule(cts, x, *, axis):
  y_bar, = cts
  return [broadcast(y_bar, x.aval.shape, axis)]
transpose_rules[reduce_sum_p] = reduce_sum_transpose_rule

def xla_call_transpose_rule(cts, *invals, jaxpr, num_consts):
  del num_consts  # Unused
  undef_primals = [type(x) is UndefPrimal for x in invals]
  transposed_jaxpr, new_consts = transpose_jaxpr(jaxpr, tuple(undef_primals))
  residuals, _ = partition_list(undef_primals, invals)
  outs = bind(xla_call_p, *new_consts, *residuals, *cts,
              jaxpr=transposed_jaxpr, num_consts=len(new_consts))
  outs = iter(outs)
  return [next(outs) if undef else None for undef in undef_primals]
transpose_rules[xla_call_p] = xla_call_transpose_rule

@lru_cache()
def transpose_jaxpr(jaxpr: Jaxpr, undef_primals: Tuple[bool, ...]
                    ) -> Tuple[Jaxpr, List[Any]]:
  avals_in, avals_out = typecheck_jaxpr(jaxpr)
  traceable = partial(eval_jaxpr_transposed, jaxpr)
  args = [UndefPrimal(a) if u else a for a, u in zip(avals_in,
undef_primals)]
  trans_jaxpr, consts, _ = make_jaxpr(traceable, tuple(args),
tuple(avals_out))
  typecheck_jaxpr(trans_jaxpr)
  return trans_jaxpr, consts
```

Now that we can linearize and transpose, we can finally write `grad`:

```
def grad(f):
  def gradfun(x, *xs):
    y, f_vjp = vjp(f, x, *xs)
    if np.shape(y) != (): raise TypeError
    x_bar, *_ = f_vjp(np.ones(np.shape(y), np.result_type(y)))
    return x_bar
  return gradfun
```

```
y, f_vjp = vjp(sin, 3.)
print(f_vjp(1.), cos(3.))
```

```
(-0.9899924966004454,) -0.9899924966004454
```

```
def f(x):
  y = sin(x) * 2.
  z = - y + x
  return z

print(grad(f)(3.))
```

```
2.979984993200891
```

```
@jit
def f(x):
  y = x * 2.
  z = g(y)
  return z

@jit
def g(x):
  return cos(x) * 2.

print(grad(f)(3.))
```

```
1.1176619927957034
```

Here’s something of a compositionality stress test:

```python
# from core_test.py fun_with_nested_calls_2
def foo(x):
  @jit
  def bar(y):
    def baz(w):
      q = jit(lambda x: y)(x)
      q = q + jit(lambda: y)()
      q = q + jit(lambda y: w + y)(y)
      q = jit(lambda w: jit(sin)(x) * y)(1.0) + q
      return q
    p, t = jvp(baz, (x + 1.0,), (y,))
    return t + (x * p)
  return bar(x)

def assert_allclose(*vals):
  for v1, v2 in zip(vals[:-1], vals[1:]):
    np.testing.assert_allclose(v1, v2)

ans1 = f(3.)
ans2 = jit(f)(3.)
ans3, _ = jvp(f, (3.,), (5.,))
ans4, _ = jvp(jit(f), (3.,), (5.,))
assert_allclose(ans1, ans2, ans3, ans4)

deriv1 = grad(f)(3.)
deriv2 = grad(jit(f))(3.)
deriv3 = jit(grad(jit(f)))(3.)
_, deriv4 = jvp(f, (3.,), (1.,))
_, deriv5 = jvp(jit(f), (3.,), (1.,))
assert_allclose(deriv1, deriv2, deriv3, deriv4, deriv5)

hess1 = grad(grad(f))(3.)
hess2 = grad(grad(jit(f)))(3.)
hess3 = grad(jit(grad(f)))(3.)
hess4 = jit(grad(grad(f)))(3.)
_, hess5 = jvp(grad(f), (3.,), (1.,))
_, hess6 = jvp(jit(grad(f)), (3.,), (1.,))
_, hess7 = jvp(jit(grad(f)), (3.,), (1.,))
assert_allclose(hess1, hess2, hess3, hess4, hess5, hess6, hess7)
```

Part 5: the control flow primitives `cond`

Next we'll add higher-order primitives for staged-out control flow. These resemble `jit` from Part 3, another higher-order primitive, but differ in that they are parameterized by multiple callables rather than just one.

Adding `cond`

We introduce a `cond` primitive to represent conditional application of one function or another inside a jaxpr. We write the type of `cond` as `Bool -> (a -> b) -> (a -> b) -> a -> b`. In words, `cond` takes a boolean representing the predicate and two functions of equal types.

Depending on the value of the predicate, it applies one function or the other to its final argument.

In Python, we represent it as a function which itself takes two functions as arguments. As with `jit`, the first step is to call `make_jaxpr` on its callable arguments to turn them into jaxprs:

```
def cond(pred, true_fn, false_fn, *operands):
  avals_in = [raise_to_shaped(get_aval(x)) for x in operands]
  true_jaxpr, true_consts, out_tree = make_jaxpr(true_fn, *avals_in)
  false_jaxpr, false_consts, out_tree_ = make_jaxpr(false_fn, *avals_in)
  if out_tree != out_tree_: raise TypeError
  true_jaxpr, false_jaxpr = _join_jaxpr_consts(
      true_jaxpr, false_jaxpr, len(true_consts), len(false_consts))
  if typecheck_jaxpr(true_jaxpr) != typecheck_jaxpr(false_jaxpr):
    raise TypeError
  outs = bind_cond(pred, *true_consts, *false_consts, *operands,
                   true_jaxpr=true_jaxpr, false_jaxpr=false_jaxpr)
  return tree_unflatten(out_tree, outs)
cond_p = Primitive('cond')

def _join_jaxpr_consts(jaxpr1: Jaxpr, jaxpr2: Jaxpr, n1: int, n2: int
                       ) -> Tuple[Jaxpr, Jaxpr]:
  jaxpr1_type, jaxpr2_type = typecheck_jaxpr(jaxpr1), typecheck_jaxpr(jaxpr2)
  assert jaxpr1_type.in_types[n1:] == jaxpr2_type.in_types[n2:]
  consts1, rest1 = split_list(jaxpr1.in_binders, n1)
  consts2, rest2 = split_list(jaxpr2.in_binders, n2)
  new_jaxpr1 = Jaxpr(consts1 + consts2 + rest1, jaxpr1.eqns, jaxpr1.outs)
  new_jaxpr2 = Jaxpr(consts1 + consts2 + rest2, jaxpr2.eqns, jaxpr2.outs)
  return new_jaxpr1, new_jaxpr2

def bind_cond(pred, *args, true_jaxpr, false_jaxpr):
  assert len(args) == len(true_jaxpr.in_binders) ==
len(false_jaxpr.in_binders)
  return bind(cond_p, pred, *args, true_jaxpr=true_jaxpr,
false_jaxpr=false_jaxpr)
```

We require `true_jaxpr` and `false_jaxpr` to have the same type, but because they might close over different constants (and because jaxprs can only represent closed terms, i.e. can't have free variables and are instead closure-converted) we need to use the helper `_join_jaxpr_consts` to make consistent the input binder lists of the two jaxprs. (To be more economical we could try to identify pairs of constants with the same shapes, but instead we just concatenate the lists of constants.)

Next we can turn to adding interpreter rules for `cond`. Its evaluation rule is simple:

```
def cond_impl(pred, *operands, true_jaxpr, false_jaxpr):
  if pred:
    return eval_jaxpr(true_jaxpr, operands)
  else:
    return eval_jaxpr(false_jaxpr, operands)
impl_rules[cond_p] = cond_impl
```

```
out = cond(True, lambda: 3, lambda: 4)
print(out)
```

```
3
```

For its JVP and vmap rules, we only need to call the same `jvp_jaxpr` and `vmap_jaxpr` utilities we created for `jit`, followed by another pass of `_join_jaxpr_consts`:

```
def cond_jvp_rule(primals, tangents, *, true_jaxpr, false_jaxpr):
  pred, *primals = primals
  _   , *tangents = tangents
  true_jaxpr , true_consts  = jvp_jaxpr(true_jaxpr)
  false_jaxpr, false_consts = jvp_jaxpr(false_jaxpr)
  true_jaxpr, false_jaxpr = _join_jaxpr_consts(
      true_jaxpr, false_jaxpr, len(true_consts), len(false_consts))
  assert typecheck_jaxpr(true_jaxpr) == typecheck_jaxpr(false_jaxpr)
  outs = bind_cond(pred, *true_consts, *false_consts, *primals, *tangents,
                   true_jaxpr=true_jaxpr, false_jaxpr=false_jaxpr)
  primals_out, tangents_out = split_half(outs)
  return primals_out, tangents_out
jvp_rules[cond_p] = cond_jvp_rule
```

```
out, out_tan = jvp(lambda x: cond(True, lambda: x * x, lambda: 0.), (1.,),
(1.,))
print(out_tan)
```

```
2.0
```

```
def cond_vmap_rule(axis_size, vals_in, dims_in, *, true_jaxpr, false_jaxpr):
  pred    , *vals_in = vals_in
  pred_dim, *dims_in = dims_in
  if pred_dim is not not_mapped: raise NotImplementedError  # TODO
  true_jaxpr, true_consts = vmap_jaxpr(true_jaxpr, axis_size, tuple(dims_in))
  false_jaxpr, false_consts = vmap_jaxpr(false_jaxpr, axis_size,
tuple(dims_in))
  true_jaxpr, false_jaxpr = _join_jaxpr_consts(
      true_jaxpr, false_jaxpr, len(true_consts), len(false_consts))
  assert typecheck_jaxpr(true_jaxpr) == typecheck_jaxpr(false_jaxpr)
  outs = bind_cond(pred, *true_consts, *false_consts, *vals_in,
                   true_jaxpr=true_jaxpr, false_jaxpr=false_jaxpr)
  return outs, [0] * len(outs)
vmap_rules[cond_p] = cond_vmap_rule
```

```
xs = np.array([1., 2., 3])
out = vmap(lambda x: cond(True, lambda: x + 1., lambda: 0.), (0,))(xs)
print(out)
```

```
[2. 3. 4.]
```

Notice that we're not currently supporting the case where the predicate value itself is batched. In mainline JAX, we handle this case by transforming the conditional to a [select primitive](). That transformation is semantically correct so long as `true_fun` and `false_fun` do not involve any side-effecting primitives.

Another thing not represented here, but present in the mainline JAX, is that applying transformations to two jaxprs of equal type might result in jaxprs of different types. For example, applying the mainline JAX version of `vmap_jaxpr` to the identity-function jaxpr

```
{ lambda a:float32[] .
  let
  in ( a ) }
```

would result in a jaxpr with a batched output, of type `[float32[10]] -> [float32[10]]` if the batch size were 10, while applying it to the zero-function jaxpr

```
{ lambda a:float32[] .
  let
  in ( 0. ) }
```

would result in a jaxpr with an unbatched output, of type `[float32[10]] -> [float32[]]`. This is an optimization, aimed at not batching values unnecessarily. But it means that in `cond` we'd need an extra step of joining the two transformed jaxprs to have consistent output types. We don't need this step here because we chose `vmap_jaxpr` always to batch all outputs over the leading axis.

Next we can turn to abstract evaluation and XLA lowering rules:

```python
def cond_abstract_eval(pred_type, *in_types, true_jaxpr, false_jaxpr):
  if pred_type != ShapedArray((), np.dtype('bool')): raise TypeError
  jaxpr_type = typecheck_jaxpr(true_jaxpr)
  if jaxpr_type != typecheck_jaxpr(false_jaxpr):
    raise TypeError
  if not all(t1 == t2 for t1, t2 in zip(jaxpr_type.in_types, in_types)):
    raise TypeError
  return jaxpr_type.out_types
abstract_eval_rules[cond_p] = cond_abstract_eval

def cond_translation(c, in_avals, in_vals, *, true_jaxpr, false_jaxpr):
  del in_avals  # Unused
  pred, *in_vals = in_vals
  flat_vals, in_tree = tree_flatten(in_vals)
  operand = xops.Tuple(c, flat_vals)
  operand_shape = c.get_shape(operand)

  def make_comp(name: str, jaxpr: Jaxpr) -> xe.XlaComputation:
    c = xc.XlaBuilder(name)
    operand = xops.Parameter(c, 0, operand_shape)
    operands = tree_unflatten(in_tree, destructure_tuple(c, operand))
    outs = jaxpr_subcomp(c, jaxpr, operands)
    return c.build(xops.Tuple(c, outs))

  true_comp = make_comp('true_fn', true_jaxpr)
  false_comp = make_comp('false_fn', false_jaxpr)

  int_etype = xc.dtype_to_etype(np.dtype('int32'))
  out = xops.Conditional(xops.ConvertElementType(pred, int_etype),
                         [false_comp, true_comp], [operand] * 2)
  return destructure_tuple(c, out)
xla_translations[cond_p] = cond_translation
```

```python
out = jit(lambda: cond(False, lambda: 1, lambda: 2))()
print(out)
```

```
2
```

Finally, to support reverse-mode automatic differentiation, we need partial evaluation and transposition rules. For partial evaluation, we need to introduce another jaxpr-munging utility, `_join_jaxpr_res`, to handle the fact that applying partial evaluation to `true_fun` and `false_fun` will in general result in distinct residuals. We use `_join_jaxpr_res` to make the output types of the transformed jaxprs consistent (while `_join_jaxpr_consts` dealt with input types).

```python
def cond_partial_eval(trace, tracers, *, true_jaxpr, false_jaxpr):
  pred_tracer, *tracers = tracers
  assert pred_tracer.pval.is_known
  pred = pred_tracer.pval.const
  in_uks = [not t.pval.is_known for t in tracers]

  *jaxprs, out_uks, num_res = _cond_partial_eval(true_jaxpr, false_jaxpr,
in_uks)
  t_jaxpr1, f_jaxpr1, t_jaxpr2, f_jaxpr2 = jaxprs

  known_tracers, unknown_tracers = partition_list(in_uks, tracers)
  known_vals = [t.pval.const for t in known_tracers]
  outs1_res = bind_cond(pred, *known_vals,
                        true_jaxpr=t_jaxpr1, false_jaxpr=f_jaxpr1)
  outs1, res = split_list(outs1_res, len(outs1_res) - num_res)
  pred_tracer_ = trace.instantiate_const(full_raise(trace, pred_tracer))
  res_tracers = [trace.instantiate_const(full_raise(trace, x)) for x in res]
  outs2 = [PartialEvalTracer(trace, PartialVal.unknown(v.aval), None)
           for v in t_jaxpr2.outs]
  eqn = JaxprEqnRecipe(cond_p, [pred_tracer_, *res_tracers,
*unknown_tracers],
                       dict(true_jaxpr=t_jaxpr2, false_jaxpr=f_jaxpr2),
                       [v.aval for v in t_jaxpr2.outs], map(ref, outs2))
  for t in outs2: t.recipe = eqn
  return merge_lists(out_uks, outs1, outs2)
partial_eval_rules[cond_p] = cond_partial_eval

def _cond_partial_eval(true_jaxpr: Jaxpr, false_jaxpr: Jaxpr, in_uks:
List[bool]
                       ) -> Tuple[Jaxpr, Jaxpr, Jaxpr, Jaxpr, List[bool],
int]:
  _, _, t_out_uks, _ = partial_eval_jaxpr(true_jaxpr , in_uks)
  _, _, f_out_uks, _ = partial_eval_jaxpr(false_jaxpr, in_uks)
  out_uks = map(op.or_, t_out_uks, f_out_uks)

  t_jaxpr1, t_jaxpr2, _, t_nres = partial_eval_jaxpr(true_jaxpr , in_uks,
out_uks)
  f_jaxpr1, f_jaxpr2, _, f_nres = partial_eval_jaxpr(false_jaxpr, in_uks,
out_uks)

  t_jaxpr1, f_jaxpr1 = _join_jaxpr_res(t_jaxpr1, f_jaxpr1, t_nres, f_nres)
  t_jaxpr2, f_jaxpr2 = _join_jaxpr_consts(t_jaxpr2, f_jaxpr2, t_nres, f_nres)
  assert typecheck_jaxpr(t_jaxpr1) == typecheck_jaxpr(f_jaxpr1)
  assert typecheck_jaxpr(t_jaxpr2) == typecheck_jaxpr(f_jaxpr2)
  num_res = t_nres + f_nres

  return t_jaxpr1, f_jaxpr1, t_jaxpr2, f_jaxpr2, out_uks, num_res

def _join_jaxpr_res(jaxpr1: Jaxpr, jaxpr2: Jaxpr, n1: int, n2: int
                    ) -> Tuple[Jaxpr, Jaxpr]:
  jaxpr1_type, jaxpr2_type = typecheck_jaxpr(jaxpr1), typecheck_jaxpr(jaxpr2)
  out_types1, _ = split_list(jaxpr1_type.out_types, len(jaxpr1.outs) - n1)
  out_types2, _ = split_list(jaxpr2_type.out_types, len(jaxpr2.outs) - n2)
  assert out_types1 == out_types2
  outs1, res1 = split_list(jaxpr1.outs, len(jaxpr1.outs) - n1)
  outs2, res2 = split_list(jaxpr2.outs, len(jaxpr2.outs) - n2)
  zeros_like1 = [Lit(np.zeros(v.aval.shape, v.aval.dtype)) for v in res1]
  zeros_like2 = [Lit(np.zeros(v.aval.shape, v.aval.dtype)) for v in res2]
  new_jaxpr1 = Jaxpr(jaxpr1.in_binders, jaxpr1.eqns, outs1 + res1 +
zeros_like2)
  new_jaxpr2 = Jaxpr(jaxpr2.in_binders, jaxpr2.eqns, outs2 + zeros_like1 +
```

```
res2)
  return new_jaxpr1, new_jaxpr2
```

```
_, f_lin = linearize(lambda x: cond(True, lambda: x, lambda: 0.), 1.)
out = f_lin(3.14)
print(out)
```

```
3.14
```

```
def cond_peval_eqn(unks_in: List[bool], eqn: JaxprEqn,
                   ) -> Tuple[JaxprEqn, JaxprEqn, List[bool], List[Atom]]:
  pred_unk, *unks_in = unks_in
  assert not pred_unk
  true_jaxpr, false_jaxpr = eqn.params['true_jaxpr'],
eqn.params['false_jaxpr']
  *jaxprs, unks_out, num_res = _cond_partial_eval(true_jaxpr, false_jaxpr,
unks_in)
  t_jaxpr1, f_jaxpr1, t_jaxpr2, f_jaxpr2 = jaxprs
  ins1, ins2 = partition_list(unks_in, eqn.inputs[1:])
  outs1, outs2 = partition_list(unks_out, eqn.out_binders)
  residuals, _ = split_list(t_jaxpr2.in_binders, num_res)
  eqn1 = JaxprEqn(cond_p, [eqn.inputs[0], *ins1],
                  dict(true_jaxpr=t_jaxpr1, false_jaxpr=f_jaxpr1),
                  outs1 + residuals)
  eqn2 = JaxprEqn(cond_p, [eqn.inputs[0], *residuals, *ins2],
                  dict(true_jaxpr=t_jaxpr2, false_jaxpr=f_jaxpr2),
                  outs2)
  res = [eqn.inputs[0], *residuals] if type(eqn.inputs[0]) is Var else
residuals
  return eqn1, eqn2, unks_out, res
partial_eval_jaxpr_rules[cond_p] = cond_peval_eqn
```

```
_, f_lin = linearize(jit(lambda x: cond(True, lambda: x, lambda: 0.)), 1.)
out = f_lin(3.14)
print(out)
```

```
3.14
```

Transposition is a fairly straightforward application of `transpose_jaxpr`:

```
def cond_transpose_rule(cts, pred, *invals, true_jaxpr, false_jaxpr):
  undef_primals = tuple(type(x) is UndefPrimal for x in invals)
  true_jaxpr, true_consts = transpose_jaxpr(true_jaxpr, undef_primals)
  false_jaxpr, false_consts = transpose_jaxpr(false_jaxpr, undef_primals)
  true_jaxpr, false_jaxpr = _join_jaxpr_consts(
      true_jaxpr, false_jaxpr, len(true_consts), len(false_consts))
  res = [x for x in invals if type(x) is not UndefPrimal]
  outs = bind_cond(pred, *true_consts, *false_consts, *res, *cts,
                   true_jaxpr=true_jaxpr, false_jaxpr=false_jaxpr)
  outs = iter(outs)
  return [None] + [next(outs) if type(x) is UndefPrimal else None for x in
invals]
transpose_rules[cond_p] = cond_transpose_rule
```

```python
out = grad(lambda x: cond(True, lambda: x * x, lambda: 0.))(1.)
print(out)
```

```
2.0
```

▼ Hide code cell source

```python
def pprint_cond(names: DefaultDict[Var, str], eqn: JaxprEqn) -> PPrint:
  true_jaxpr, false_jaxpr = eqn.params['true_jaxpr'],
eqn.params['false_jaxpr']
  new_params = {k:v for k, v in eqn.params.items() if not
k.endswith('jaxpr')}
  lhs = pp(' '.join(var_str(names, v) for v in eqn.out_binders))
  rhs = (pp(eqn.primitive.name) >> pp_params(new_params) >>
         pp(' '.join(names[x] if isinstance(x, Var) else str(x.val)
                     for x in eqn.inputs)))
  return vcat([lhs >> pp(' = ') >> rhs,
               pp_jaxpr(true_jaxpr).indent(2),
               pp_jaxpr(false_jaxpr).indent(2)])
pp_rules[cond_p] = pprint_cond
```