Chopper

Release 0.0.1

Enflame

Dec 01, 2021

CONTENTS

1 Pass And Transforms **1**
- 1.1 CANCERConversionPasses . . . 1
- 1.2 RefBackendPasses . . . 1
- 1.3 Transforms . . . 2
- 1.4 CANCERNumpyTransforms . . . 2
- 1.5 CANCERTCFTransforms . . . 2
- 1.6 CANCERTCPTransforms . . . 2

2 Basicpy **3**
- 2.1 BasicpyDialect . . . 3
- 2.2 BasicpyOps . . . 3

3 Numpy **15**
- 3.1 NumpyOps . . . 15
- 3.2 NumpyDialect . . . 20

4 cancer_frontend **21**
- 4.1 cancer_frontend package . . . 21

Python Module Index **25**

CHAPTER

ONE

PASS AND TRANSFORMS

1.1 CANCERConversionPasses

1.1.1 `-convert-basicpy-to-std`: Convert representable Basicpy ops to std

1.1.2 `-convert-numpy-to-tcf`: Convert the numpy dialect to supported TCF ops

1.1.3 `-convert-tcf-to-linalg`: Convert TCF to Linalg

The intention is for this pass to convert mainly to linalg named ops.

Because linalg is at the "TCP" layer of abstraction, this pass has to concern itself with generating guards for error cases. ### `-convert-tcf-to-std`: Convert TCF to Std ### `-convert-tcf-to-tcp`: Convert TCF to TCP

1.2 RefBackendPasses

1.2.1 `-lower-alloc-memref-ops`: Lower AllocMemRefOp's

1.2.2 `-lower-to-refbackrt-abi`: Lower constructs requiring runtime support to `refbackrt`

We have a specialized dialect `refbackrt` which models our runtime's data structures, and function signatures (and presumably eventually, other ABI boundaries like external calls if we ever support it) will be converted.

The constructs requiring runtime support are: - function signatures / module metadata - error handling ### `-refback-lower-to-llvm`: Lower everything to LLVM ### `-restricted-canonicalize`: Canonicalize operations This pass is the same as the regular `canonicalize` pass, but it only applies a restricted set of patterns.

This is useful when a particular canonicalization is actually needed for correctness of a lowering flow. For such cases, running a restricted set of canonicalizations makes it clearer which passes are needed for correctness and which passes are "just optimizations". This helps when debugging miscompiles and other situations where the compiler is not behaving as expected.

Options

```
-included-dialects : Which dialects should be canonicalized
```

1.3 Transforms

1.3.1 `-basicpy-type-inference`: Performs function level type inference

1.4 CANCERNumpyTransforms

1.4.1 `-numpy-array-to-tensor`: Replace arrays with tensors where possible (optimization only).

This pass is analogous to an SSA-formation pass in a traditional compiler, with the added complication that arrays can alias each other in interesting ways.

The current code doesn't implement any fancy algorithm, and is intended to be just sufficient for a first e2e spike. An algorithm inspired by the SSA formation literature will need to be implemented.

Also, this pass doesn't currently handle interprocedural rewriting (of private functions), which is even more complex. ### `-numpy-public-functions-to-tensor`: Converts public functions to operate on tensors (instead of ndarray) ### `-numpy-refine-public-return`: Refine public return Refines types of values return from public functions based on intraprocedural information.

This pass effectively encodes an assumption by the pass pipeline author that the public calling convention of the module can have its types refined, without causing ABI mismatches. This is frequently true – for example, in many systems, `tensor<?x?xf32>`, `tensor<3x3xf32>` and `tensor<*x!numpy.any_dtype>` are all the same data structure on calling convention boundaries.

This pass is expected to run after shape refinement has occurred to otherwise resolve shapes, and is currently mainly useful to convert rank/dtype-erased function boundaries to ranked, dtyped code for compiler backends.

1.5 CANCERTCFTransforms

1.5.1 `-tcf-shape-refinement`: Refines shapes of tensors

1.6 CANCERTCPTransforms

1.6.1 `-tcp-bufferize`: Bufferizes the tcp dialect

CHAPTER

TWO

BASICPY

2.1 BasicpyDialect

2.2 BasicpyOps

2.2.1 `basicpy.as_i1` (::mlir::CANCER::Basicpy::AsI1Op)

Evaluates an input to an i1 predicate value

Syntax:

```
operation ::= `basicpy.as_i1` $operand attr-dict `:` type($operand)
```

Applies the rules for interpreting a type as a boolean, returning an i1 indicating the truthiness of the operand. Since the output of this op is intended to drive lower-level control flow, the i1 type is used (not the user level BoolType).

Operands:

Operand	Description
`operand`	any type

Results:

Result	Description
`result`	1-bit signless integer

2.2.2 `basicpy.binary_compare` (::mlir::CANCER::Basicpy::BinaryCompareOp)

Performs a comparison between two operands

Syntax:

```
operation ::= `basicpy.binary_compare` $left $operation $right attr-dict `:`␣
↪type(operands)
```

This op performs only one step of a potentially multi-step short circuit comparison. See: https://docs.python.org/3/reference/expressions.html#comparisons

Attributes:

Attribute	MLIR Type	Description
`operation`	::mlir::StringAttr	Comparison operator

Operands:

Operand	Description
`left`	any type
`right`	any type

Results:

Result	Description
`result`	Bool type

2.2.3 `basicpy.binary_expr` (::mlir::CANCER::Basicpy::BinaryExprOp)

Binary expression

Syntax:

```
operation ::= `basicpy.binary_expr` $left $operation $right attr-dict `:` functional-
↪type(operands, results)
```

An expression between two operands as generated by the AST BinOp node.

Attributes:

Attribute	MLIR Type	Description
`operation`	::mlir::StringAttr	Operation for a binary expression

Operands:

Operand	Description
`left`	any type
`right`	any type

Results:

Result	Description
`result`	any type

2.2.4 `basicpy.bool_cast` (::mlir::CANCER::Basicpy::BoolCastOp)

Casts between BoolType and i1 (predicate value)

Syntax:

```
operation ::= `basicpy.bool_cast` $operand attr-dict `:` type(operands) `->`␣
↪type(results)
```

When interfacing with lower level dialect or progressively lowering the Python BoolType away, it is often necessary to cast between it and i1, which is used to represent bool-ness at lower levels.

Operands:

Operand	Description
`operand`	Python bool or i1

Results:

Result	Description
`result`	Python bool or i1

2.2.5 `basicpy.bool_constant` (::mlir::CANCER::Basicpy::BoolConstantOp)

A boolean constant

Syntax:

```
operation ::= `basicpy.bool_constant` $value attr-dict
```

A constant of type !basicpy.BoolType that can take either an i1 value of 0 (False) or 1 (True).

Note that as in Python a BoolType can be thought of as an object, whereas the corresponding i1 is a numeric type suitable for use in contexts where storage format matters (or for interop with lower level dialects).

Attributes:

Attribute	MLIR Type	Description
`value`	::mlir::IntegerAttr	1-bit signless integer attribute

Results:

Result	Description
`result`	Bool type

2.2.6 `basicpy.build_dict` (::mlir::CANCER::Basicpy::BuildDictOp)

Builds an empty dict

Syntax:

```
operation ::= `basicpy.build_dict` attr-dict `:` functional-type(operands, results)
```

This op mirrors the CPython BUILD_MAP op (note naming difference).

Note that as with CPython, this op only builds an empty dict; however, it is reserved in the future for it to take variadic operands to construct with a list of key/value pairs.

Results:

Result	Description
`result`	Dict type

2.2.7 `basicpy.build_list` (::mlir::CANCER::Basicpy::BuildListOp)

Builds a list from operands

Syntax:

```
operation ::= `basicpy.build_list` operands attr-dict `:` functional-type(operands,␣
↪results)
```

Constructs a new list object from its operands.

TODO: Any allowable type can be expressed in lists; however, this should be revisited once more of the dialect infrastructure is in place and tightened up accordingly. At that time, appropriate constraints should be added that both allow correct program representation and support transformations to lower levels (i.e. allowing a wider set of types as useful for conversions).

Operands:

Operand	Description
`elements`	any type

Results:

Result	Description
`result`	List type

2.2.8 `basicpy.build_tuple` (::mlir::CANCER::Basicpy::BuildTupleOp)

Builds a tuple from operands

Syntax:

```
operation ::= `basicpy.build_tuple` operands attr-dict `:` functional-type(operands,␣
↪results)
```

Constructs a new tuple object from its operands.

TODO: Any allowable type can be expressed in lists; however, this should be revisited once more of the dialect infrastructure is in place and tightened up accordingly. At that time, appropriate constraints should be added that both allow correct program representation and support transformations to lower levels (i.e. allowing a wider set of types as useful for conversions).

Operands:

Operand	Description
`elements`	any type

Results:

Result	Description
`result`	Tuple type

2.2.9 `basicpy.bytes_constant` (::mlir::CANCER::Basicpy::BytesConstantOp)

Constant bytes value

Syntax:

```
operation ::= `basicpy.bytes_constant` $value attr-dict
```

A bytes value of BytesType. The value is represented by a StringAttr.

Attributes:

Attribute	MLIR Type	Description
`value`	::mlir::StringAttr	string attribute

Results:

Result	Description
`result`	Bytes type

2.2.10 `basicpy.exec_discard` (::mlir::CANCER::Basicpy::ExecDiscardOp)

Terminator for an exec block

Syntax:

```
operation ::= `basicpy.exec_discard` operands attr-dict `:` type(operands)
```

Discards results and terminates an exec block.

Operands:

Operand	Description
`operands`	any type

2.2.11 `basicpy.exec` (::mlir::CANCER::Basicpy::ExecOp)

Evaluates an expression being executed as a statement

The result is discarded. Typically expressions are no-side-effect and can be re-ordered as needed. Embedding one in an exec op ensures that its placement in program order is preserved.

2.2.12 `basicpy.func_template_call` (::mlir::CANCER::Basicpy::FuncTemplateCallOp)

Calls a function template

Syntax:

```
operation ::= `basicpy.func_template_call` $callee `(` $args `)` `kw` $arg_names attr-
↪dict `:` functional-type($args, results)
```

Most function calls start with this generic calling op, which binds symbolically to a func_template. At this level, there are very few semantics associated with the call, since, often, both types and the specific concrete callee cannot be determined.

Per python calling conventions, all functions return one result, even if None or a tuple (which may be syntactically unpacked to multiple results).

If specified, the `argNames` operand is right aligned to the list of positional `args`, representing arguments that are special or have been passed with a keyword. The following arg names are special: '*': Indicates that the argument is a positional argument pack (must be the first arg name, if present). '**': Indicates that the argument is a keyword argument pack (must be the last arg name, if present).

Attributes:

Attribute	MLIR Type	Description
`callee`	::mlir::FlatSymbolRefAttr	flat symbol reference attribute
`arg_names`	::mlir::ArrayAttr	string array attribute

Operands:

Operand	Description
`args`	any type

Results:

Result	Description
`result`	any type

2.2.13 `basicpy.func_template` (::mlir::CANCER::Basicpy::FuncTemplateOp)

Group of multiple overload-resolved concrete functions

The outer func_template op acts as a module that can contain named concrete functions that are interpreted as overloads. If the function signature is sufficient to disambiguate (i.e. with nothing more than arity and MLIR argument types), then this is all that is needed. However, in many cases, additional attributes will need to be specified to further constrain types. The first matching function signature is selected to satisfy a `func_template_call` op.

TODO: Define this extended constraint matching.

Once a concrete function is selected as being applicable to a given call, it will typically be instantiated as a standalone, unspecialized function in the calling module (as a peer to the func_template). This function will be uniquely identified by concating the outer func_template's symbol name, '$', and the concrete instance's symbol name.

Note that the function may still be unspecialized (in that it contains UnknownType arguments/results), and type inference is expected to further specialize/inline/constrain it.

By convention, func_templates are named to avoid collision for various uses: - Global function templates: "**global:math:`python.qualified.name" - Method names: "__method`method_name" - Attribute getter:** "getattrattr$_name" - Attributesetter : "_{setattrattr\_name}$

As in user-level python, for functions that bind to an instance, the first argument must be a concrete type for the bound instance type. In this way, there is one `func_template` for every unique member name and the normal type constraints system is used to select the overload, just as if it was a normal function call. It is left to utility routines to merge libraries in a way that preserves this invariant.

TODO: This needs to be fleshed out more as some additional rules about ordering and conflict resolution are likely needed to make this correct.

When extracting a program, it is typically necessary to create weak references to specific python functions and correlate them back to a named template defined here. Often times this can just be done lexically, but to avoid fragility, any func_template that correlates to a python runtime function will have an additional attribute `py_bind` that is an array of StringAttr qualified names to resolve and bind to in the python runtime. In cases of divergence, the symbol name of the template should be chosen just for uniqueness (not significance).

The qualified name format for `py_bind` attribute is: package.name#local.qualified.name

2.2.14 `basicpy.func_template_terminator` (::mlir::CANCER::Basicpy::FuncTemplateTerminator

Terminator pseudo-op for the FuncTemplateOp

2.2.15 `basicpy.numeric_constant` (::mlir::CANCER::Basicpy::NumericConstantOp)

A constant from the Python3 numeric type hierarchy

Basicpy re-uses core MLIR types to represent the Python3 numeric type hierarchy with the following mappings:

- Python3 `int` : In python, this type is signed, arbitrary precision but in typical realizations, it maps to an MLIR `IntegerType` of a fixed bit-width (typically si64 if no further information is known). In the future, there may be a real `Basicpy::IntType` that retains the true arbitrary precision nature, but this is deemed an enhancement that does not obviate the need to infer physical, sized types for many real-world cases. As such, the Basicpy numeric type hierarchy will always include physical `IntegerType`, if only to enable progressive lowering and interop with cases where the precise type is known.
- Python3 `float` : This is allowed to map to any legal floating point type on the physical machine and is usually represented as a double (f64). In MLIR, any `FloatType` is allowed, which facilitates progressive lowering and interop with cases where a more precise type is known.
- Python3 `complex` : Maps to an MLIR `ComplexType` with a `FloatType` elementType (note: in Python, complex numbers are always defined with floating point components).
- `bool` : See `bool_constant` for a constant (i1) -> !basicpy.BoolType constant. This constant op is not used for representing such bool values, even though from the Python perspective, bool is part of the numeric hierarchy (the distinction is really only necessary during promotion).

2.2.16 Integer Signedness

All `int` values in Python are signed. However, there exist special cases where libraries (i.e. struct packing and numpy arrays) interoperate with unsigned values. As such, when mapping to MLIR, Python integer types are represented as either signed or unsigned `IntegerType` types and can be lowered to signless integers as appropriate (typically during realization of arithmetic expressions where the choice is meaningful). Since it is not known at the outset when in lowering this information is safe to discard this `numeric_constant` op accepts any signedness.

Attributes:

Attribute	MLIR Type	Description
`value`	::mlir::Attribute	any attribute

Results:

Result	Description
«unnamed»	any type

2.2.17 `basicpy.singleton` (::mlir::CANCER::Basicpy::SingletonOp)

Constant value for a singleton type

Syntax:

```
operation ::= `basicpy.singleton` attr-dict `:` type($result)
```

Some types only have a single possible value, represented by the SingletonAttr. This op allows creating constants of these types.

Results:

Result	Description
`result`	None type or Ellipsis type

2.2.18 `basicpy.slot_object_get` (::mlir::CANCER::Basicpy::SlotObjectGetOp)

Gets a slot from a slot object

Gets a slot from a SlotObject.

Example: %0 = basicpy.slot_object_make … %1 = basicpy.slot_object_get %0[1] : !basicpy.SlotObject<…>

Attributes:

Attribute	MLIR Type	Description
`index`	::mlir::IntegerAttr	index attribute

Operands:

Operand	Description
`object`	Slot object

Results:

Result	Description
`result`	any type

2.2.19 `basicpy.slot_object_make` (::mlir::CANCER::Basicpy::SlotObjectMakeOp)

Creates an instance of a SlotObject type

SlotObjects are typically instances of built-in classes that have a fixed number of slots. Unlike in standard python, the types of each slot are tracked.

This op has a custom assembly form which can be used when valid that omits the operand types (since they are equal to the types in the returned slot object). Example: %0 = basicpy.singleton : !basicpy.NoneType %1 = basicpy.slot_object_make(%0) -> !basicpy.SlotObject

Operands:

Operand	Description
`slots`	any type

Results:

Result	Description
`result`	Slot object

2.2.20 `basicpy.str_constant` (::mlir::CANCER::Basicpy::StrConstantOp)

Constant string value

Syntax:

```
operation ::= `basicpy.str_constant` $value attr-dict
```

A string value of StrType. The value is represented by a StringAttr that is UTF-8 encoded.

Attributes:

Attribute	MLIR Type	Description
`value`	::mlir::StringAttr	string attribute

Results:

Result	Description
`result`	String type

2.2.21 `basicpy.unknown_cast` (::mlir::CANCER::Basicpy::UnknownCastOp)

Casts to and from the UnknownType

Syntax:

```
operation ::= `basicpy.unknown_cast` operands attr-dict `:` type(operands) `->`␣
↪type(results)
```

Operands:

Operand	Description
`operand`	any type

Results:

Result	Description
`result`	any type

CHAPTER THREE

NUMPY

3.1 NumpyOps

3.1.1 `numpy.builtin_ufunc_call` (::mlir::CANCER::Numpy::BuiltinUfuncCallOp)

A **call** operation on a named/builtin ufunc

Syntax:

```
operation ::= `numpy.builtin_ufunc_call` `<` $qualified_name `>` `(` operands `)` attr-
↪dict `:` functional-type(operands, results)
```

Simple ufunc call semantics for builtin ufuncs with none of the advanced arguments specified.

Note that without the `out=` parameter, ufunc call operations (unlike others like `at`) are defined purely in the value domain and do not alias. As such, they operate on tensors, not ndarray.

Attributes:

Attribute	MLIR Type	Description
`qualified_name`	::mlir::StringAttr	string attribute

Operands:

Operand	Description
`inputs`	tensor of any type values

Results:

Result	Description
`output`	tensor of any type values

3.1.2 `numpy.copy_to_tensor` (::mlir::CANCER::Numpy::CopyToTensorOp)

Copies an ndarray, yielding a value-typed tensor.

Syntax:

```
operation ::= `numpy.copy_to_tensor` $source attr-dict `:` functional-type($source,
↪$dest)
```

The semantics of this operation connote a copy of the data in the source ndarray, producing a destination value that will have the value in the ndarray at the point of the copy. Of course, downstream transformations are free to rearrange things to elide the copy or otherwise eliminate the need for it.

Operands:

Operand	Description
`source`	ndarray type

Results:

Result	Description
`dest`	tensor of any type values

3.1.3 `numpy.create_array_from_tensor` (::mlir::CANCER::Numpy::CreateArrayFromTensorOp)

Creates an ndarray from a tensor.

Syntax:

```
operation ::= `numpy.create_array_from_tensor` $source attr-dict `:` functional-type(
↪$source, $dest)
```

Creates a new ndarray that will contain the data of the given tensor.

Operands:

Operand	Description
`source`	tensor of any type values

Results:

Result	Description
`dest`	tensor of any type values or ndarray type

3.1.4 `numpy.dot` (::mlir::CANCER::Numpy::DotOp)

Represents the `numpy.dot` operator

Syntax:

```
operation ::= `numpy.dot` operands attr-dict `:` functional-type(operands, $output)
```

See: https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html

Operands:

Operand	Description
a	tensor of any type values or ndarray type
b	tensor of any type values or ndarray type

Results:

Result	Description
`output`	tensor of any type values or ndarray type

3.1.5 `numpy.get_slice` (::mlir::CANCER::Numpy::GetSliceOp)

Gets a slice of an array

Syntax:

```
operation ::= `numpy.get_slice` operands attr-dict `:` functional-type(operands, $result)
```

This op encapsulates all forms of indexing into an array by taking a variable number of `slice` arguments, each of which represents a single entry in a generalized indexing-tuple. Once full type inference has been performed, there should be sufficient static information to determine the exact slice semantics solely by the signature of types of the `slice` arguments.

Note that there is a more general form of this op that is generally needed for AST extraction that takes a variable length `tuple` instead of a static list of arguments. It is expected that during type refinement most such uses should degenerate to this static variant.

Per numpy semantics, many forms of slice return a view instead of a copy, and determining the exact form requires additional analysis.

Operands:

Operand	Description
a	tensor of any type values or ndarray type
`slice_elements`	types that are legal elements of a **getitem** tuple operating on arrays

Results:

Result	Description
`result`	tensor of any type values or ndarray type

3.1.6 `numpy.narrow` (::mlir::CANCER::Numpy::NarrowOp)

Narrows an array to a known type at boundaries.

Syntax:

```
operation ::= `numpy.narrow` $operand attr-dict `:` functional-type($operand, $result)
```

During tracing, specific data types are often unknown. This op generically narrows from an unknown to a known data type at boundaries.

Operands:

Operand	Description
`operand`	tensor of any type values or ndarray type

Results:

Result	Description
`result`	tensor of any type values or ndarray type

3.1.7 `numpy.overwrite_array` (::mlir::CANCER::Numpy::OverwriteArrayOp)

Ovewrite the contents of array with a tensor.

Syntax:

```
operation ::= `numpy.overwrite_array` $tensor `overwrites` $array attr-dict `:` type(
↪$tensor) `,` type($array)
```

Replaces the contents of `array` with corresponding values from `tensor`.

Immediately after this op has completed, indexing `array` will result in identical values as indexing into `tensor`. Of course, later ops might mutate `array`, so this relationship need not hold for the entire program.

This op has undefined behavior if the tensor and array have different shapes or dtypes.

Operands:

Operand	Description
`tensor`	tensor of any type values
`array`	ndarray type

3.1.8 `numpy.static_info_cast` (::mlir::CANCER::Numpy::StaticInfoCastOp)

Adds/removes static information from an array type.

Syntax:

```
operation ::= `numpy.static_info_cast` $operand attr-dict `:` type($operand) `to` type(
↪$result)
```

This op does not imply any runtime code. Semantically it is an identity function.

Operands:

Operand	Description
`operand`	tensor of any type values or ndarray type

Results:

Result	Description
`result`	tensor of any type values or ndarray type

3.1.9 `numpy.tensor_static_info_cast` (::mlir::CANCER::Numpy::TensorStaticInfoCastOp)

Adds/removes static information from a tensor type.

Syntax:

```
operation ::= `numpy.tensor_static_info_cast` $operand attr-dict `:` type($operand) `to`␣
↪type($result)
```

This op does not imply any runtime code. Semantically it is an identity function.

Unlike `tensor.cast`, this op allows changing dtype, following the rules of numpy arrays where no runtime code is implied. In particular, `!numpy.any_dtype` is compatible with all other element types, but otherwise the element types must be the same. An element type of `!numpy.any_dtype` represents the absence of static knowledge of the dtype. It does not itself represent a concrete runtime element type.

Operands:

Operand	Description
`operand`	tensor of any type values

Results:

Result	Description
`result`	tensor of any type values

3.1.10 `numpy.transpose` (::mlir::CANCER::Numpy::TransposeOp)

Represents the `numpy.transpose` op with no permutation specified

Syntax:

```
operation ::= `numpy.transpose` operands attr-dict `:` functional-type(operands, $output)
```

This op is equivalent to calling `numpy.transpose(arr)`, which reverses the axes of the array. It is separate from the explicit form because it is not always possible to locallly infer an appropriate axis transform at the point of declaration.

See: https://docs.scipy.org/doc/numpy/reference/generated/numpy.transpose.html

Operands:

Operand	Description
`a`	tensor of any type values or ndarray type

Results:

Result	Description
`output`	tensor of any type values or ndarray type

3.2 NumpyDialect

CHAPTER

FOUR

CANCER_FRONTEND

4.1 cancer_frontend package

4.1.1 Subpackages

cancer_frontend.numpy package

Module contents

cancer_frontend.python package

Submodules

cancer_frontend.python.python_jit_runner module

Contains jit runner class for compilation and execution

`class cancer_frontend.python.python_jit_runner.`**`PythonRunner`**

Bases: `object`

PythonRunner class that is a compiler supports jit functionalities for numpy DSL.

Returns returns the instance of this class

Return type *PythonRunner*

`dump_mlir``(`*_ast: mlir.astnodes.Node*`)` → str

`dump_python``(`*_ast: _ast.AST*`)` → str

`parse_mlir``(`*code_path: str*`)` → mlir.astnodes.Node

Parses the code by providing its path :param

`parse_python``(`*func: Callable*`)` → _ast.AST

Parses the code by providing its path :param

Module contents

cancer_frontend.scaffold package

Subpackages

cancer_frontend.scaffold.mlir_dialects package

Submodules

cancer_frontend.scaffold.mlir_dialects.dialect_demo module

Dialect classes that create and custom dialect as example.

cancer_frontend.scaffold.mlir_dialects.dialect_pynative module

Implemented classes of NativePython Dialect.

cancer_frontend.scaffold.mlir_dialects.dialect_tcf module

Implemented classes of Tensor Computation Flow Dialect.

Module contents

cancer_frontend.scaffold.utils package

Submodules

cancer_frontend.scaffold.utils.class_utils module

`class cancer_frontend.scaffold.utils.class_utils.`**`classproperty`**`(`*f*`)`

Bases: `object`

@classmethod+@property

`class cancer_frontend.scaffold.utils.class_utils.`**`memoized_classproperty`**`(`*f*`)`

Bases: `object`

@classmethod+@property

cancer_frontend.scaffold.utils.display_util module

class cancer_frontend.scaffold.utils.display_util.**ColorPalette**

Bases: `object`

ENDC = '\x1b[0m'

FAIL = '\x1b[91m'

HEADER = '\x1b[95m'

WARNING = '\x1b[93m'

cancer_frontend.scaffold.utils.file_util module

cancer_frontend.scaffold.utils.file_util.**dump_to_file**()

cancer_frontend.scaffold.utils.file_util.**read_src**(*filename: basestring*) → basestring

cancer_frontend.scaffold.utils.import_util module

cancer_frontend.scaffold.utils.import_util.**get_python_library**()

cancer_frontend.scaffold.utils.import_util.**get_python_methods**(*module*)

cancer_frontend.scaffold.utils.import_util.**wraps**(*wrapped, assigned=('__module__', '__name__', '__qualname__', '__doc__', '__annotations__'), updated=('__dict__',)*)

Decorator factory to apply update_wrapper() to a wrapper function

Returns a decorator that invokes update_wrapper() with the decorated function as the wrapper argument and the arguments to wraps() as the remaining arguments. Default arguments are as for update_wrapper(). This is a convenience function to simplify applying partial() to update_wrapper().

cancer_frontend.scaffold.utils.json_parser module

Simple JSON Parser

The code is short and clear, and outperforms every other parser (that's written in Python). For an explanation, check out the JSON parser tutorial at /docs/json_tutorial.md (this is here for use by the other examples)

class cancer_frontend.scaffold.utils.json_parser.**TreeToJson**(*visit_tokens=True*)

Bases: `lark.visitors.Transformer`

array

alias of `list`

false(_)

null(_)

number = <function float>

object

alias of `dict`

pair

alias of `tuple`

`string(s)`

`true(_)`

cancer_frontend.scaffold.utils.json_parser_test module

Module contents

Module contents

cancer_frontend.torch package

Module contents

4.1.2 Module contents

PYTHON MODULE INDEX

C

cancer_frontend, 24
cancer_frontend.python, 22
cancer_frontend.python.python_jit_runner, 21
cancer_frontend.scaffold, 24
cancer_frontend.scaffold.mlir_dialects, 22
cancer_frontend.scaffold.mlir_dialects.dialect_demo, 22
cancer_frontend.scaffold.mlir_dialects.dialect_pynative, 22
cancer_frontend.scaffold.mlir_dialects.dialect_tcf, 22
cancer_frontend.scaffold.utils, 24
cancer_frontend.scaffold.utils.class_utils, 22
cancer_frontend.scaffold.utils.display_util, 23
cancer_frontend.scaffold.utils.file_util, 23
cancer_frontend.scaffold.utils.import_util, 23
cancer_frontend.scaffold.utils.json_parser, 23
cancer_frontend.torch, 24

INDEX

A

array (*cancer_frontend.scaffold.utils.json_parser.TreeToJson attribute*), 23

C

cancer_frontend
    module, 24
cancer_frontend.python
    module, 22
cancer_frontend.python.python_jit_runner
    module, 21
cancer_frontend.scaffold
    module, 24
cancer_frontend.scaffold.mlir_dialects
    module, 22
cancer_frontend.scaffold.mlir_dialects.dialect_demo
    module, 22
cancer_frontend.scaffold.mlir_dialects.dialect_pynative
    module, 22
cancer_frontend.scaffold.mlir_dialects.dialect_tcf
    module, 22
cancer_frontend.scaffold.utils
    module, 24
cancer_frontend.scaffold.utils.class_utils
    module, 22
cancer_frontend.scaffold.utils.display_util
    module, 23
cancer_frontend.scaffold.utils.file_util
    module, 23
cancer_frontend.scaffold.utils.import_util
    module, 23
cancer_frontend.scaffold.utils.json_parser
    module, 23
cancer_frontend.torch
    module, 24
classproperty (*class in cancer_frontend.scaffold.utils.class_utils*), 22
ColorPalette (*class in cancer_frontend.scaffold.utils.display_util*), 23

D

dump_mlir() (*cancer_frontend.python.python_jit_runner.PythonRunner method*), 21
dump_python() (*cancer_frontend.python.python_jit_runner.PythonRunner method*), 21
dump_to_file() (*in module cancer_frontend.scaffold.utils.file_util*), 23

E

ENDC (*cancer_frontend.scaffold.utils.display_util.ColorPalette attribute*), 23

F

FAIL (*cancer_frontend.scaffold.utils.display_util.ColorPalette attribute*), 23
false() (*cancer_frontend.scaffold.utils.json_parser.TreeToJson method*), 23

G

get_python_library() (*in module cancer_frontend.scaffold.utils.import_util*), 23
get_python_methods() (*in module cancer_frontend.scaffold.utils.import_util*), 23

H

HEADER (*cancer_frontend.scaffold.utils.display_util.ColorPalette attribute*), 23

M

memoized_classproperty (*class in cancer_frontend.scaffold.utils.class_utils*), 22
module
    cancer_frontend, 24
    cancer_frontend.python, 22
    cancer_frontend.python.python_jit_runner, 21
    cancer_frontend.scaffold, 24
    cancer_frontend.scaffold.mlir_dialects, 22
    cancer_frontend.scaffold.mlir_dialects.dialect_demo, 22

cancer_frontend.scaffold.mlir_dialects.dialect_pynative, 22
cancer_frontend.scaffold.mlir_dialects.dialect_tcf, 22
cancer_frontend.scaffold.utils, 24
cancer_frontend.scaffold.utils.class_utils, 22
cancer_frontend.scaffold.utils.display_util, 23
cancer_frontend.scaffold.utils.file_util, 23
cancer_frontend.scaffold.utils.import_util, 23
cancer_frontend.scaffold.utils.json_parser, 23
cancer_frontend.torch, 24

N

null() (*cancer_frontend.scaffold.utils.json_parser.TreeToJson method*), 23
number (*cancer_frontend.scaffold.utils.json_parser.TreeToJson attribute*), 23

O

object (*cancer_frontend.scaffold.utils.json_parser.TreeToJson attribute*), 23

P

pair (*cancer_frontend.scaffold.utils.json_parser.TreeToJson attribute*), 23
parse_mlir() (*cancer_frontend.python.python_jit_runner.PythonRunner method*), 21
parse_python() (*cancer_frontend.python.python_jit_runner.PythonRunner method*), 21
PythonRunner (*class in cancer_frontend.python.python_jit_runner*), 21

R

read_src() (*in module cancer_frontend.scaffold.utils.file_util*), 23

S

string() (*cancer_frontend.scaffold.utils.json_parser.TreeToJson method*), 23

T

TreeToJson (*class in cancer_frontend.scaffold.utils.json_parser*), 23
true() (*cancer_frontend.scaffold.utils.json_parser.TreeToJson method*), 24

W

WARNING (*cancer_frontend.scaffold.utils.display_util.ColorPalette attribute*), 23
wraps() (*in module cancer_frontend.scaffold.utils.import_util*), 23