CSCS

Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Eve Workshop

Simple definition and transformation of tree-like Intermediate Representations in Python

Day 1: 28.10.2020
Day 2: ??

Motivation

- Requirements for a suitable toolchain technology
  - Simple and productive prototyping (*"focus on the problem, not the framework"*)
    - Minimal boilerplate
    - Easy to learn, easy to use, easy to debug
  - Stable and reliable enough to be used in production if needed
  - Performance is not a concern
  - Mostly AST/tree transformation (*"our toolchains generate code, not Assembly"*)
    - **GT4Py**: Python AST -> GTScript AST -> GTIR -> … -> Python/C++/CUDA code
    - **Dawn**: C++AST -> GTClang AST -> SIR -> IIR -> C++/CUDA code
- Reuse interesting features and designs from previous experiences
  - Python ***ast***: declarative node definitions (Zephyr ASDL), tree visitors
  - **MLIR**: validation, dialects, human-readable serialization
  - Text templating engines: powerful and readable code generation

Motivation: decision

- *"Let's use Python!"*
  - A high-productivity environment with great REPL tools
  - Easy to learn
  - Easy to connect to other tools and languages
  - Powerful built-in meta-programming features

- *"Let's extract and enhance current GT4Py IR utils!"*
  - A light-weight set of utils to simplify the work we **actually** do with IRs

- *"Let's focus only on the infrastructure!"*
  - Share basic tools among **unrelated** toolchains or products
  - Decouple **design** of IRs and optimizations from **implementation**

So… what is Eve?

- A light-weight Python framework to define and work with tree-like IRs
  - Define node classes
  - Automatic (and custom) validation of node data
  - Transformation of node trees
  - Code generation

and what is Eve not?

- GT4Py
- A compiler toolchain for weather and climate applications
  - **GTC** is a prototype toolchain using infrastructure provided by Eve
- A set of pre-defined IRs
- Magic

Design concepts: Node

- ***Nodes*** are the minimal **units** of IRs
  - A node defines a specific **concept** of the IR domain
- **Data-only** objects
  - Nodes contains values/data
  - Decouple transformations from the specific data representation
  - Easy to serialize / deserialize
- Supported attributes
  - Built-in types (*int*, *float*, *str*)
  - Collections (*List*, *Dict*, *Set*) and *Enums*
  - *Dataclasses*-like (***pydantic*** models)
  - Other node classes (trees)
- ☑ 👍 Tip: Treat node as **immutable** objects (GTC approach)

Node definition: classes in Python

- **Goal**: declarative node definition
  - Python Zephyr ASDL
  - MLIR ODS (*TableGen*)
  - GT4Py (custom *attribclass*) 😄

- **Problem***:* Python objects are dynamic
  - Hard to define *structs* with fixed fields
  - Recurrent problem in Python
    - *slots*
    - *namedtuples*
    - *dataclasses*
    - …
  - Validation of type annotations is not enforced at run-time
    - only for linting tools

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? returns,
                         string? type_comment)

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list)
      | Return(expr? value)
```

```
class ClassDef:
    def __init__(self, name, bases, keywords, body, decorator_list):
        # Verbose init function
        self.name = name
        self.bases = bases
        self.keywords = keywords
        self.body = body
        self.decorator_list = decorator_list

# No type checking
node = ClassDef("name", [], [], ..., [])

# Regulars instances are dicts, not structs
node.whatever = "This also works"
```

Node definition: dataclasses syntax and Pydantic features

- Definition: *dataclasses*-like syntax
- Validation: data-validation frameworks
  - *attrs (used in GT4Py)*
  - ***pydantic***
  - *typic.al*
  - …

- ***pydantic*** comes with a rich API and a convenient set of features
  - type validation (although **not** strict)
  - per-field validators and root validators (pre and post)
  - frozen (immutable) classes
  - automatic generation of JSON schema

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? returns,
                         string? type_comment)

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list)
      | Return(expr? value)
```

```
class FunctionDef(Node):
    name: Str
    args: Arguments
    body: List[Stmt]
    decorator_list: List[Expr]
    returns: Optional[Expr]
    type_comment: Optional[Str]


class ClassDef(Node):
    name: Str
    bases: List[Str]
    keywords: List[Keyword]
    body: List[Stmt]
    decorator_list: List[Expr]
```

Node definition: example

- Each field is declared with type annotations:
  - Field_name: type
- Type is checked on instance creation

  BinaryOperator(
  left=valid_expr, op=”+”, right=valid_expr
) # ok

  BinaryOperator(
  left=”bla”, op=”+”, right=valid_expr
) # raise Exception(‘left’ is not a valid expr)

- Additional validators are also supported

```
class Expr(Node):
    location_type: Optional[LocationType]
    loc: Optional[SourceLocation]

class BinaryOperator(Expr):
    left: Expr
    op: Str
    right: Expr

class Expr(Node):
    location_type: Optional[LocationType]
    loc: Optional[SourceLocation]

class BinaryOperator(Expr):
    left: Expr
    op: Str
    right: Expr

class FieldAccessExpr(Expr):
    name: Str
    vertical_offset: Int
    horizontal_offset: Union[CartesianOffset,
        UnstructuredOffset, ZeroOffset]
```

Node implementation: details

- What is exactly a **Node**?
  - A dataclass-like Python class with *fields*
    - *Currently* implemented as a Pydantic Model
  - Extra iterators to visit children
  - MRO bases (custom metaclass):
    1. *eve.BaseNode*
    2. *pydantic.BaseModel*
    3. *abc.ABC*

- What is exactly a **field**?
  - Instance member generated by dataclass-like frameworks
  - Declared as a class member with a type annotation
  - Usually comes with extra features (e.g. validation)

- What is a **dialect**?
  - Set of Nodes defined inside a Python module
  - It does not need to define a full grammar

```python
class LocationType(Enum):
    VERTEX: "vertex"
    EDGE: "edge"

class Expr(Node):
    location_type: Optional[LocationType]
    loc: Optional[SourceLocation]

class BinaryOperator(Expr):
    left: Expr
    op: str
    right: Expr

bin_op = BinaryOperator(
    left=Expr(), op="+", right=Expr()
)
for name, field in bin_op.iter_children():
    print(name, type(field))


location_type <class 'NoneType'>
loc <class 'NoneType'>
left <class '__main__.Expr'>
op <class 'str'>
right <class '__main__.Expr'>
```

Node implementation: naming conventions

- Names starting with **"_"** (e.g. _*field*) are ignored by **pydantic** & **Eve** (☑ 👍 Tip: **Don't** use)
- Names ending with **"__"** (e.g. *internal__*) are reserved for internal use (☑ 👍 Tip: **Don't** use)
- Field names ending with **"_"** are considered **implementation fields**
  - Not visible in regular children iterator
  - Typically used to cache derived, non-essential information on the node
- Any other field not starting or ending with *underscore* is a regular field

```python
# Node definition
class MyNode(Node):
    __mangled = ...     # Name mangled class attribute, ignored by pydantic and Eve
    _hidden = ...       # Regular class attribute, ignored by pydantic and Eve
    internal__: Any     # Regular pydantic field, ignored by Eve (internal use)
    impl_: Any          # Regular pydantic field, implementation field for Eve
    field: Str          # Regular node field

assert MyNode._MyNode__mangled == ...
assert MyNode._hidden == ...

my_node = MyNode(internal_="any", impl_="impl_value", field="field_value")
assert my_node.internal__ == "any"
assert my_node.impl_ == "impl_value"
assert my_node.field == "field"_value

assert [name for name, _ in my_node.iter_children()] == ["field"]
assert [name for name, _ in my_node.iter_impl_fields()] == ["impl_"]
```

Node validation

- Custom validators are defined with decorators
  - *@validator(“field_name”)*
  - *@root_validator()*
  - Custom logic in the validator
  - Raise errors when it is wrong

- *Pre-validators*
  - pre=True
  - executed **before** type validation

```
class AssignStmt(Node):
    left: FieldAccess
    right: Expr

    @validator('left')
    def no_offset_in_assignment_lhs(cls, v):
        if v.offset.i != 0 or v.offset.j != 0:
            raise ValueError(
                'Lhs of assignment must not have an offset'
            )
        return v
```

Code generation: dumping IRs as strings

- **TemplatedGenerator** is a special visitor using text templates to *render* nodes

- Define a template for each **Node** class
  - format()
  - String.Template
  - Mako
  - Jinja2

- If additional processing is needed
  - Add custom *visit_method(...)*

- Eve provides source code formatting
  - Python (*black*)
  - C++ (*clang-format*)

```
class NaiveCodeGenerator(codegen.TemplatedGenerator):
    ...

    BinaryOp = as_fmt("{left} {op} {right}")
    ExprStmt = as_fmt("\n{expr};")
    VarDeclStmt = as_fmt("\n{data_type} {name};")
```

```
formatted = codegen.format_source(
    "cpp", code, style="LLVM"
)
```

Code generation: example

Example: part of *NaiveCodeGenerator*

```
from eve.codegen import FormatTemplate as as_fmt
from eve.codegen import MakoTemplate as as_mako

class NaiveCodeGenerator(codegen.TemplatedGenerator):
    ...

    AssignmentExpr = as_fmt("{left} = {right}")
    VarAccessExpr = as_fmt("{name}")
    BinaryOp = as_fmt("{left} {op} {right}")
    ExprStmt = as_fmt("\n{expr};")
    VarDeclStmt = as_fmt("\n{data_type} {name};")

    ...

    UnstructuredField = as_mako(
"""<%
loc_type = location_type["singular"]
sparseloc = "sparse_" if _this_node.sparse_location_type else ""
%>
dawn::${ sparseloc }${ loc_type }_field_t<LibTag, ${ data_type }>& ${ name };"""
    )
```

Code generation: templates

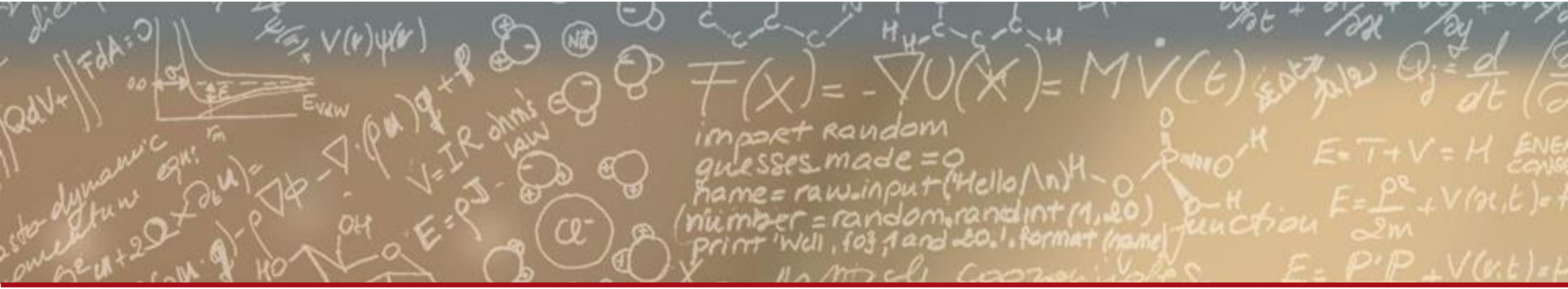
- Templates are class variables in a *TemplatedGenerator* class:
  - Variable name is the Node name
  - Variable content is a *eve.codegen.Template* instance

- Available variables in templates
  - `**node_fields` => all the children and implementation fields by name
  - `_impl: Dict[str, Any]` => results of visiting all the node implementation fields
  - `_children: Dict[str, Any]` => results of visiting all the node children
  - `_this_node: Node` => actual node instance (before visiting children)
  - `_this_generator: TemplateGenerator` => the current generator instance
  - `_this_module: Module` => the generator's module instance
  - `_kwargs: Dict[str, Any]` => keyword arguments received by the visiting method

Code generation: visitors

- Custom *visit_method(...)* can be combined with regular templates
  - *visit_method()* does some preprocessing
  - Calls to *generic_visit()* triggers normal template processing

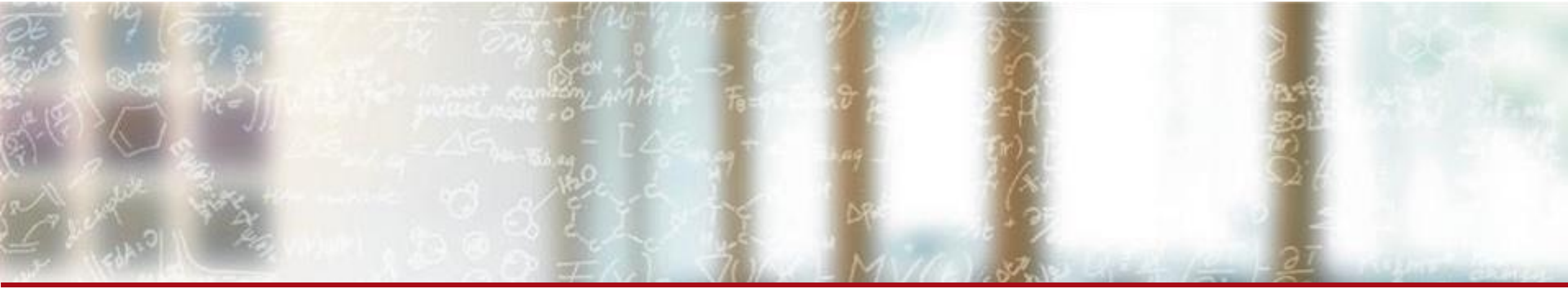
```
class NaiveCodeGenerator(codegen.TemplatedGenerator):

ReduceOverNeighbourExpr = as_mako("""<%
    ...
%>""")

...

def visit_ReduceOverNeighbourExpr(self, node, *, iter_var, **kwargs) -> str:
    outer_iter_var = iter_var
    return self.generic_visit(
        node, outer_iter_var=outer_iter_var, iter_var="redIdx", **kwargs
    )
```



Thanks for your attention

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Eve Framework

Hands-on session: https://deepnote.com/project/bc7a9798-a9c7-47ca-a989-93f266a17705