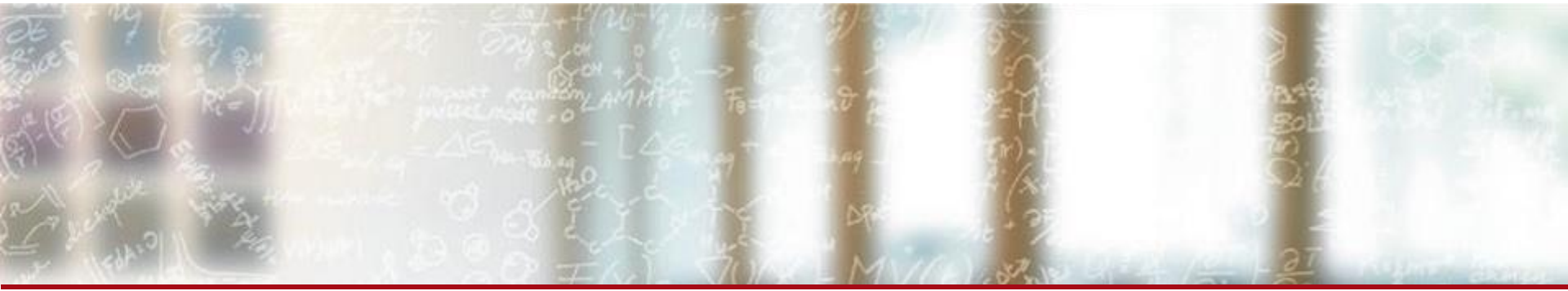


Eve Workshop

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

Day 1: 28.10.2020
Day 2: 30.10.2020

Design concepts: Visitor

- **Visitors** implement operations involving node trees
  - Follow pattern in the standard Python **ast** module (*visit_CLASS_NAME*)
- ***NodeVisitor***
  - Read-only operations on node trees (☑👍Tip: Treat node as **immutable** objects)
  - Default behavior: nothing
- ***NodeTranslator***
  - Translate an input node tree into a new output tree
  - Default behavior: deepcopy of the input tree
- Stateless or stateful data collection visitors? Both approaches work
  - Stateless visitor can pass around information
    - To children: using *kwargs*
    - To parents: using *return* values
  - Stateful visitors can define internal data structures to hold specific data
  - Extra info can be attached to the tree nodes can using *implementation fields*

Visitors and Translators

- Method naming:
  - *visit_CLASS_NAME()*
  - Problem: typos in the method name!!
- Dispatching algorithm

```
def visit(Node):
    visit_name = type(node).__name__
    if hasattr(Visitor, visit_name):
        Visitor.visit_name(node, **kwargs)
    elif Node is eve.Node:
            # try with node.__mro__ bases
    # If nothing has found...
    Visitor.generic_visit(node, **kwargs)
```

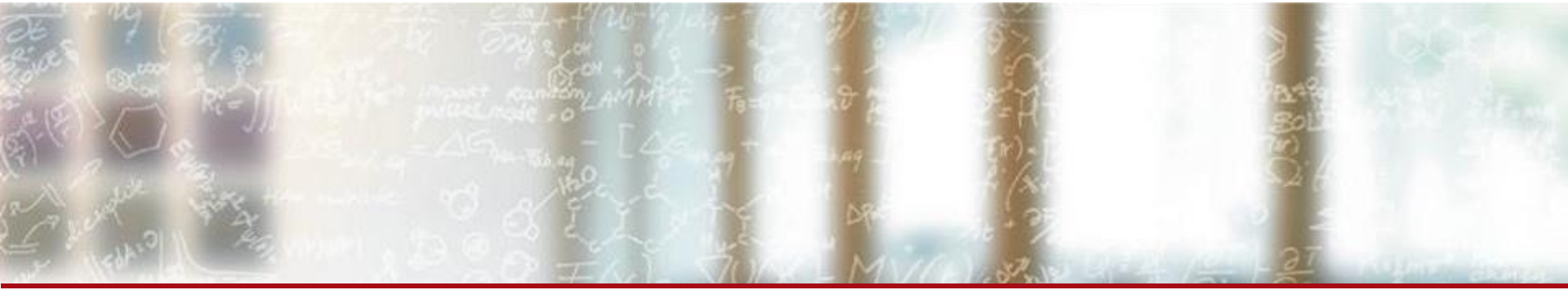
- **Translator** return values
  - New node or ***eve.NOTHING***
- *kwargs* are forwarded by default

```python
class GtirToNir(eve.NodeTranslator):
    ...

def visit_FieldAccess(self, node: gtir.FieldAccess, *, locs,
**kwargs):
    ...

    return nir.FieldAccess(
        name=node.name,
        location_type=node.location_type,
        primary=primary_chain,
        secondary=secondary_chain,
    )

def visit_NeighborReduce(self, node: gtir.NeighborReduce,
**kwargs):
    ...
    return nir.VarAccess(...)

def visit_Literal(self, node: gtir.Literal, **kwargs):
    for

def visit_BinaryOp(self, node: gtir.BinaryOp, **kwargs):
    ...

def visit_AssignStmt(self, node: gtir.AssignStmt, **kwargs):
    ...
```

Status and outlook

- Current state
  - IR definitions
  - Analysis and transformations
  - Code generation

- Work in progress / experimental
  - Generic nodes
  - Symbol table: store information about scopes and symbols
  - Node iterators and tree navigation
  - Pass manager: cache results of analysis passes and re-run automatically when IR transformations might affect the results
  - Minor Pydantic annoyances (strict validation for nodes, hashing nodes, …)
  - Automatic checking of typos in visitor names (using *accepted_nodes* info)



Eve Framework

Hands-on session

Generic nodes (experimental / WIP feature)

- Reuse common patterns
- Generic nodes
  - Defined with ***TypeVar**s*
  - Provided by **pydantic**
- Concrete instantiations
  - Use specific nodes
  - Inherit generic validators
  - Support adding specific validators
- It works across dialects

```python
# Generic node definitions
ExprT = TypeVar("ExprT")

class GenericBinaryOp(GenericNode, Generic[ExprT]):
    op: BinaryOperator
    left: ExprT
    right: ExprT

LeftT = TypeVar("LeftT")

class GenericAssignStmt(GenericNode, Generic[LeftT, ExprT]):
    left: LeftT
    right: ExprT


# Concrete instantiations
class FieldBinaryOp(GenericBinaryOp[FieldExpr]):
    # Defining specific exta fields or validators here
    ...

class AssignStmt(GenericAssignStmt[Union[FieldAccess, VarAccess], Expr]):
    ...
```

Symbols definition (experimental / WIP feature)

- Tag node fields as symbol definitions
  - (Optional) constraining RegExp
- Tag nodes opening new scopes with the **SymbolTable** trait
  - Automatic collection of symbols in the scope

```
class LeafNode(Node):
    name: SymbolName.constrained(r"[a-zA-Z_]+")
    int_value: Int

class SimpleNode(Node):
    left: LeafNode
    right: LeafNode

# CompoundNode instances automatically collect
# all the symbols defined in any of its descendants
class CompoundNode(Node, SymbolTableTrait):
    node: SimpleNode
    value: int
```

Visitors and node paths (experimental / WIP feature)

- TreeVisitors: a new visitor class with access to the node path
  - NodePath class allows the navigating the tree upwards (and downwards)
  - Useful for Analysis passes adding information to the nodes

```
class ExtraVisitor(TreeVisitor):

    def visit_Node(self, node, node_path, **kwargs):
        print(f"Current path: {str(node_path)} (Node = {node})")
        print(f"Parent path: {str(node_path.parent)} (Parent = {node_path.parent.node}")
        grandpa = node_path[2]
        some_ancestor_node = node_path.parent.parent.parent.node
```

Node Iterators (experimental / WIP feature)

- Generators to iterate over all the tree nodes with regular statements
  - **for** loops, **itertools** and any other function accepting iterators
- Different iteration orders available (pre-order, post-order, levels-order)
- Extensible (register new operators in the **NodeIterator** class)

```
for node in traverse_tree(root_node, TraversalOrder.PRE_ORDER):
    if isinstance(node, Node):
        print(f"Node Id: {node.id_}")

collections = []
for node in traverse_tree(root_node, TraversalOrder.LEVELS_ORDER):
    if isinstance(node, collections.abc.Mapping):
        collections.append(node)


floats = traverse_tree(root_node).filter_by_type(float)
more_floats = traverse_tree(root_node).filter_by_type(int).map(lambda x: float(x))
max_scalar = max(floats.chain(more_loats.truediv(1.2232)))
```

Status and outlook

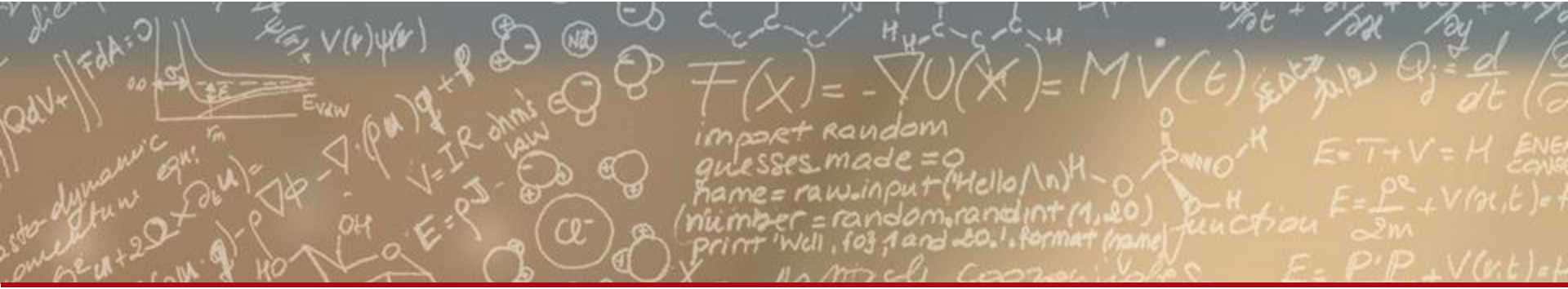
- Current state
  - IR definitions
  - Analysis and transformations
  - Code generation

- Work in progress / experimental
  - Generic nodes
  - Symbol table: store information about scopes and symbols
  - Node iterators and tree navigation
  - Pass manager: cache results of analysis passes and re-run automatically when IR transformations might affect the results
  - Minor Pydantic annoyances (strict validation for nodes, hashing nodes, …)
  - Automatic checking of typos in visitor names (using *accepted_nodes* info)

Summary

- Light-weight framework for designing self-validating IRs
- Flexible code generator with templates
- Focus on the domain and the algorithms, not implementation details or toolchain performance

  "If the algorithm is complicated, it will be complicated in any language, but with python access to helper libraries is easier, e.g. for representing graphs"
- Python makes it easy to connect and import/export data from other languages and toolchains



Thanks for your attention