Extending WLANG with references

Kabiir Krishna and Seturaj Matroja

University of Waterloo, 200 University Ave W, Waterloo, ON N2L 3G1

Abstract. This project extends the While Language (`wlang`) by introducing references, enabling advanced memory management and data manipulation capabilities. References in programming languages allow variables to be allocated on the heap and accessed indirectly, facilitating dynamic memory allocation, data sharing, and modular program design. Our implementation involves syntactic extensions for reference declaration (`ref x := new 42`) and dereferencing (`y := *x`). We modified the Abstract Syntax Tree (AST) to include new nodes for reference-related operations and updated the parser to recognize these new constructs. The interpreter was extended to manage a heap using a dictionary, enabling proper handling of references. Through rigorous testing with various test cases, we validated the functionality and correctness of our implementation. This extension enhances `wlang`'s versatility and power, making it suitable for more complex computational tasks. The project showcases our design decisions, theoretical foundations, and practical implementation challenges, culminating in a more robust and capable language for users.

Keywords: While Language · References · Memory Management · Data Manipulation · Abstract Syntax Tree (AST) · Interpreter · Heap Allocation · Syntax Extension · Dynamic Memory · Programming Language Enhancement.

1 Introduction

This project aims to extend the `wlang` (While Language) with references. A reference is a variable that is allocated on the heap, which allows for more advanced memory management and data manipulation capabilities. This extension will significantly enhance the functionality of `wlang`, making it more powerful and versatile for various computational tasks.

2 Team Member Details

2.1 Kabiir Krishna

- **Email:** k7krishn@uwaterloo.ca
- **WATIAM:** k7krishn
- **Student Number:** 21106092

2.2 Seturaj Matroja

– **Email:** smatroja@uwaterloo.ca
– **WATIAM:** smatroja
– **Student Number:** 21064444

3 Theoretical Foundations

3.1 References and Memory Management

In programming languages, a reference is an alias or a pointer to a value stored in memory. References enable indirect access to memory locations, which allows for dynamic memory management, sharing of data between different parts of a program, and efficient manipulation of complex data structures.

3.2 Benefits of References

1. **Dynamic Memory Allocation:** References allow variables to be allocated dynamically on the heap.
2. **Data Sharing:** Multiple variables can refer to the same memory location, enabling shared data manipulation.
3. **Encapsulation and Modularity:** References enable more modular program design by encapsulating data and providing controlled access.

4 Design Decisions

To introduce references, we decided on a simple and intuitive syntax which closely follows notation standards prescribed by languages like **C** or **C++** for familiarity & ease-of-use.
We also added additional functionalities & commands like AddressOf and `print_heap` so developers can easily debug their referencing-related code in-program itself.

4.1 Syntax for Reference Declaration

For declaring a reference, the keyword 'ref' is used, followed by the variable name and the value it points to:

```
ref x := new 42
```

4.2 Syntax for Dereferencing

Dereferencing a reference, i.e., accessing the value it points to, is done using the '*' operator:

```
    y := *x
```

4.3 Syntax for AddressOf (Additional Feature)

To get the memory address of a variable and then store it in the pointer variable, we employ the '&' operator.

```
    z := &y
```

4.4 Syntax for print_heap (Additional Feature)

To help debug and track heap memory utilization, we further extended the `while` language with a `print_heap` command to output the contents of the heap.

```
    print_heap
```

4.5 Abstract Syntax Tree (AST) Modifications

We extended the existing `AstVisitor`, `PrintVisitor` in `ast.py` to support new reference-related operations by adding specific nodes for reference declarations and dereferencing.

4.6 Interpreter and Parser Modifications

We modified the interpreter to manage a separate heap for references and updated the parser to recognize and correctly parse the new syntax.

5 Implementation

5.1 AST modifications

To represent reference-related operations, we added new nodes in `ast.py`. These nodes enable the AST to handle reference declarations and dereferencing appropriately.

```python
# Created a class for pointer declaration statement
# It has two attributes lhs and rhs
# lhs is the left hand side of the statement
# rhs is the right hand side of the statement

class PointerDeclStmt(Stmt):
    """A pointer declaration statement"""

    def __init__(self, lhs, rhs):
        self.lhs = lhs
        self.rhs = rhs

    def __eq__(self, other):
        return (
            type(self) == type(other)
            and self.lhs == other.lhs
            and self.rhs == other.rhs
        )

# Created a class for pointer dereference statement
# It has two attributes lhs and rhs
# lhs is the left hand side of the statement
# rhs is the right hand side of the statement

class PointerDerefStmt(Stmt):
    """A pointer dereference statement"""

    def __init__(self, lhs, rhs):
        self.lhs = lhs
        self.rhs = rhs

    def __eq__(self, other):
        return (
            type(self) == type(other)
            and self.lhs == other.lhs
            and self.rhs == other.rhs
        )

class AddressOf(Exp):
    """An address of expression"""

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

    def __eq__(self, other):
        return type(self) == type(other) and self.var ==
   other.var

# Created a class that prints the heap in-program
class PrintHeapStmt(Stmt):
    """Print Heap state"""

    def __eq__(self, other):
        return type(self) == type(other)
# ....

class AstVisitor(object):
    """Base class for AST visitor"""

        visitor = getattr(self, "visit_" + Stmt.__name__)
        return visitor(node, *args, **kwargs)


    # Generated visitor method for pointer declaration statement
    def visit_PointerDeclStmt(self, node, *args, **kwargs):
        visitor = getattr(self, "visit_" + Stmt.__name__)
        return visitor(node, *args, **kwargs)

    # Generated visitor method for pointer dereference statement
    def visit_PointerDerefStmt(self, node, *args, **kwargs):
        visitor = getattr(self, "visit_" + Stmt.__name__)
        return visitor(node, *args, **kwargs)

    # Generated visitor method for address of expression
    def visit_AddressOf(self, node, *args, **kwargs):
        visitor = getattr(self, "visit_" + Exp.__name__)
        return visitor(node, *args, **kwargs)

    # Generated visitor method for address of expression
    def visit_PrintHeapStmt(self, node, *args, **kwargs):
        visitor = getattr(self, "visit_" + Stmt.__name__)
        return visitor(node, *args, **kwargs)

class PrintVisitor(AstVisitor):
    def visit_AddressOf(self, node, *args, **kwargs):
        self._write("&")
        self.visit(node.var, *args, **kwargs)

    def visit_PointerDeclStmt(self, node, *args, **kwargs):
        self._write("ref ")
        self.visit(node.lhs)
        self._write(" := ")
        self.visit(node.rhs, no_brkt=True)

    def visit_PointerDerefStmt(self, node, *args, **kwargs):
        # self._write("pointer_deref ")
        self.visit(node.lhs)
        self._write(" := *")
        self.visit(node.rhs, no_brkt=True)

    def visit_PrintHeapStmt(self, node, *args, **kwargs):
        self._write("print_heap")
```

5.2 Parser modifications

The parser was updated to recognize and correctly parse reference declarations and dereferencing:

```
class WhileLangParser(Parser):
    # base code ...

    # Added pointer decl stmt method and generated the
   nodes for the stmt
    @tatsumasu()
    def _pointer_decl_stmt_(self):
        self._token('ref')
        self._name_()
        self.name_last_node('lhs')
        self._token(':=')
        self._aexp_()
        self.name_last_node('rhs')
        self._define(
            ['lhs', 'rhs'],
            []
        )
    # Added pointer deref stmt method and generated the
   nodes for the stmt
    @tatsumasu()
    def _pointer_deref_stmt_(self):
        self._name_()
        self.name_last_node('lhs')
        self._token(':=')
        self._token('*')
        self._name_()
        self.name_last_node('rhs')
        self._define(
            ['lhs', 'rhs'],
            []
        )

    # Added print heap stmt method to generate node for
   print_heap command
```

```
    @tatsumasu()
    def _print_heap_stmt_(self):  # noqa
        self._token('print_heap')
```

5.3 Interpreter modifications

The interpreter was extended to handle reference operations. A dictionary was used as the heap to manage references:

```
class Interpreter(ast.AstVisitor):
    def __init__(self):
        pass
        self.heap = {} # Heap is created for storing
    addresses of the references of the pointers

    def run(self, ast, state):
        return self.visit(ast, state=state)
            st.env[v.name] = 0
        return st

    # The visit method for the PointerDeclStmt and
    PointerDerefStmt is implemented
    # Here the heap is used to store the address of the
    reference of the pointer

    def visit_PointerDeclStmt(self, node, *args, **kwargs
    ):
        st = kwargs["state"]
        value = self.visit(node.rhs, *args, **kwargs)
        address = id(value)  # Use id to simulate a
    memory address
        self.heap[address] = value  # Store the value in
    the heap
        st.env[node.lhs.name] = address  # Initialize the
     pointer in the state
        print(f"Pointer declared {node.lhs.name} with
    value {value} at address {address}")
        return st

    def visit_PointerDerefStmt(self, node, *args, **
    kwargs):
        st = kwargs["state"]
        if node.rhs.name not in st.env:
            raise KeyError(f"Pointer '{node.rhs.name}' is
     not initialized in the state")

```

```
        address = st.env[node.rhs.name]
        if address not in self.heap:
            raise KeyError(f"Address '{address}' is not
    initialized in the heap")

        st.env[node.lhs.name] = self.heap[address]
        print(f"Dereferencing pointer {node.rhs.name}
    with address {address} to value {self.heap[address]}")
        return st

    def visit_AddressOf(self, node, *args, **kwargs):
        st = kwargs["state"]
        var_name = node.var.name
        if var_name not in st.env:
            raise KeyError(f"Variable '{var_name}' is not
     initialized")
        address = st.env[var_name]
        print(f"Address of {var_name}: {address}")
        return address  # Return the address of the
    variable

    def visit_PrintHeapStmt(self, node, *args, **kwargs):
        print("Heap: ", self.heap)
```

5.4 Semantics modifications

The semantics was updated to recognize and correctly return the ast for the references:

```
class WlangSemantics(object):
    def __init__(self):
        pass

    def pointer_decl_stmt(self,stmt,*args,**kwargs):
        return ast.PointerDeclStmt(stmt.lhs,stmt.rhs)

    def pointer_deref_stmt(self,stmt,*args,**kwargs):
        return ast.PointerDerefStmt(stmt.lhs,stmt.rhs)

    def address_of(self,stmt,*args,**kwargs):
        return ast.AddressOf(stmt.var)

    def print_heap_stmt(self, stmt, *args, **kwargs):
        return ast.PrintHeapStmt()
```

6 Testing

Our testing activities for this project are robust and comprehensive, as evidenced by the execution of a total of 81 test cases across multiple test scripts including test_int.py, `test_util.py`, `test_stats_visitor.py`, `test_undef_visitor.py`, `test_pointer.py`, and `test_sym.py`. Utilizing Python's `unittest` framework, our main test suite runner effectively aggregates and executes all these tests, ensuring thorough validation of the project's components. The overall test coverage of 96% indicates that the vast majority of the codebase is rigorously tested. All test cases pass successfully, which confirms the reliability and correctness of our implementation.

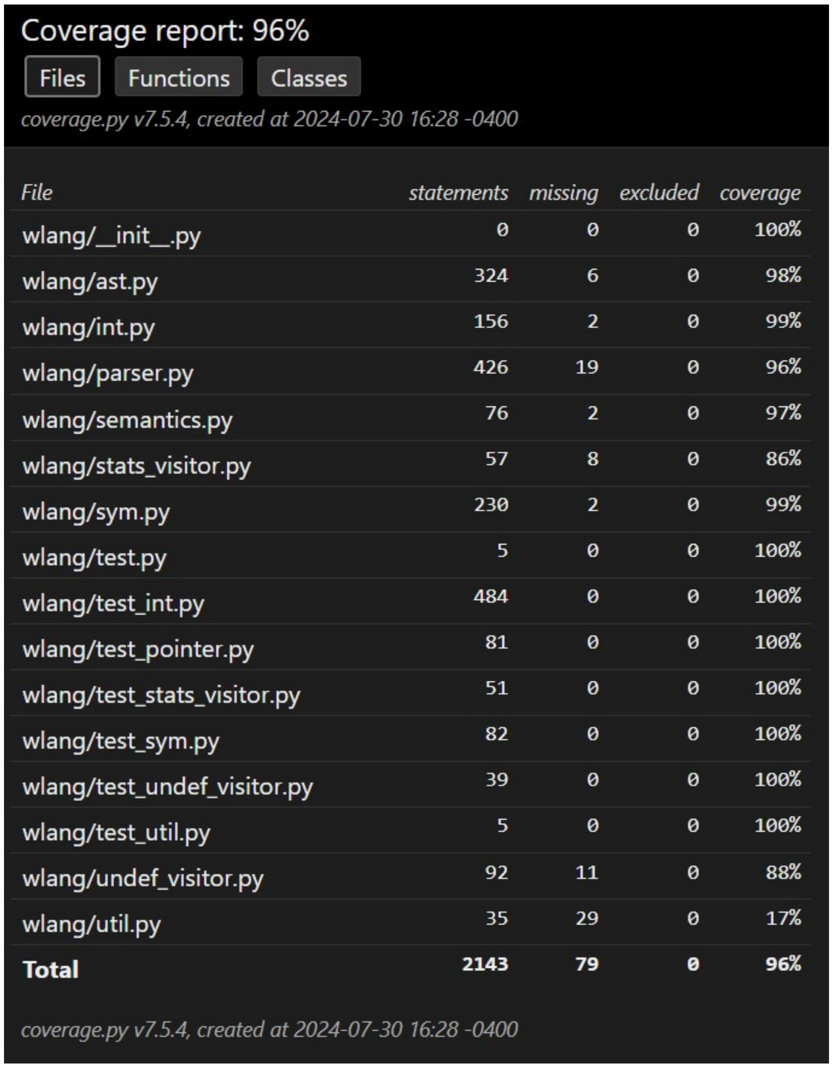
A sample test cases for testing out new reference feature is :

```
class TestInt(unittest.TestCase):
    def test_ref1(self):
        prg1 = "ref x:=42; y := *x ;print_state"
        ast1 = ast.parse_string(prg1)
        interp = int.Interpreter()
        state = int.State()
        interp.run(ast1, state)
        self.assertEquals(state.env['y'], 42)
```

```
class TestInt(unittest.TestCase):
    def test_ref2(self):
        prg1 = "ref y:=42; x := &y ; z := *x ;
    print_state"
        ast1 = ast.parse_string(prg1)
        interp = int.Interpreter()
        state = int.State()
        interp.run(ast1, state)
        self.assertEquals(state.env['z'], 42)
```

6.1 Test coverage Report

We were able to write test cases that covered the existing code, as well as our new reference feature as well, coverage for the while language is 96%. The coverage report (figure below) for the key files `ast.py`, `int.py`, `sym.py`, and `parser.py` indicates comprehensive testing with minor room for improvements



Coverage report: 96%

Files Functions Classes

coverage.py v7.5.4, created at 2024-07-30 16:28 -0400

File	*statements*	*missing*	*excluded*	*coverage*
wlang/__init__.py	0	0	0	100%
wlang/ast.py	324	6	0	98%
wlang/int.py	156	2	0	99%
wlang/parser.py	426	19	0	96%
wlang/semantics.py	76	2	0	97%
wlang/stats_visitor.py	57	8	0	86%
wlang/sym.py	230	2	0	99%
wlang/test.py	5	0	0	100%
wlang/test_int.py	484	0	0	100%
wlang/test_pointer.py	81	0	0	100%
wlang/test_stats_visitor.py	51	0	0	100%
wlang/test_sym.py	82	0	0	100%
wlang/test_undef_visitor.py	39	0	0	100%
wlang/test_util.py	5	0	0	100%
wlang/undef_visitor.py	92	11	0	88%
wlang/util.py	35	29	0	17%
Total	**2143**	**79**	**0**	**96%**

coverage.py v7.5.4, created at 2024-07-30 16:28 -0400

Fig. 1. The coverage report for the entire wlang.

7 Challenges and Solutions

1. **Memory Management**
   Solution: We implemented a simple heap management system using a dictionary to store references and their corresponding values.
2. **Syntax Integration**
   Solution: We carefully designed the new syntax for reference declaration and dereferencing to integrate seamlessly with the existing wlang syntax.

3. **AST and Parser Updates**
   Solution: We updated the AST and parser to recognize and handle the new reference-related syntax, ensuring that all cases were correctly parsed and evaluated.

8 Conclusion

By extending wlang with references, we have significantly enhanced its capabilities, allowing for more advanced memory management and data manipulation. This extension not only makes wlang more powerful but also opens up new possibilities for its use in more complex computational tasks. With these new capabilities, developers can create more efficient and robust applications, handling larger datasets and more intricate algorithms with ease. Furthermore, the introduction of references paves the way for future enhancements such as real-time debugging tools and performance optimization features, which could further elevate the utility and applicability of wlang in various high-demand computing environments.

9 Future Work

In our ongoing efforts to enhance *wlang*, a key area of development will be the implementation of memory management functionalities, specifically `malloc` and `calloc`. These features will allow users to dynamically allocate and initialize memory, significantly increasing the control they have over memory usage within their applications. Alongside this, we plan to introduce a garbage collection mechanism to efficiently manage memory and prevent potential memory leaks. This will be essential for maintaining optimal performance as the complexity of user programs increases.

10 References

References

1. ECE 653 Lecture Notes
2. Manoj Kumar Srivastav, Asoke Nath: MATHEMATICAL DESCRIPTION OF VARIABLES, POINTERS, STRUCTURES, UNIONS USED IN C-TYPE LANGUAGE. CONFERENCE Journal of Global Research in Computer Science Volume 5, No. 2, February 2014. ISSN-2229-371X