Re-implement property wrappers with macros

Candidate Info

Name: Arihant Marwaha

GitHub: https://github.com/ArihantMarwaha

Email: arihantmarwaha@icloud.com

Twitter/X: @MarwahaArihant

University Course: B.tech Computer Science

University: Amity School of Engineering and Technology, Noida

Time Zone: Noida, Uttar Pradesh (GMT+5:30)

Project Information

Organization: Swift.org

Project: Re-implement property wrappers with macros

Project size: 350 hours (large)

Estimated difficulty: Intermediate

Recommended skills: Proficiency in Swift and C++

Potential mentors: Pavel Yaskevich

Abstract:

Property wrappers in Swift are currently implemented directly in the compiler with special-case code scattered throughout the codebase. With the recent introduction of Swift macros and init accessors, we now have an opportunity to reimplement property wrappers as a combination of these standard language features, removing the need for special compiler handling.

Introduction:

Property wrappers, introduced in Swift 5.1, provide a way to abstract property-related logic such as validation, transformation, and storage. Currently, they're implemented with special handling at each stage of the compiler pipeline - parsing, semantic analysis, SIL generation, and code generation. With the introduction of Swift macros in Swift 5.9 and init accessors, we now have the foundation to reimplement property wrappers using standard language features.

This proposal outlines a detailed plan to reimplement Swift's property wrappers using macros and initializers, eliminating special-case compiler code while maintaining complete backward compatibility. The project aligns with Swift's evolution direction toward implementing language features with consistent primitives rather than special-case handling throughout the compiler pipeline.

Project goals:

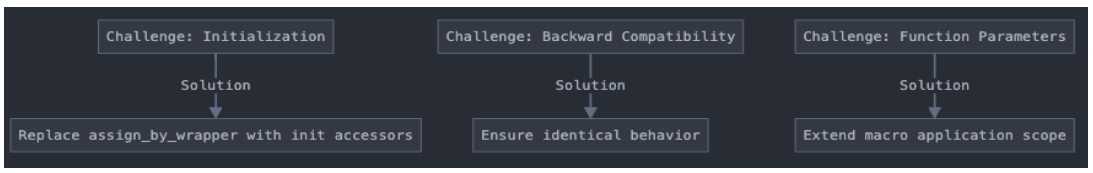
1. Remove all property wrapper-specific code from the Swift compiler
2. Ensure full backward compatibility with existing property wrapper syntax
3. Leverage Swift macros to generate the same transformations property wrappers currently do
4. Utilize init accessors for initialization instead of custom SIL instructions
5. Improve code maintainability, testability, and reduce complexity in the compiler

Technical Challenges and Solutions:

1. Ensuring Macros Have Equivalent Capabilities

Property wrappers can currently modify storage, transform access, and inject code in various contexts. Macros need to support all these transformations.

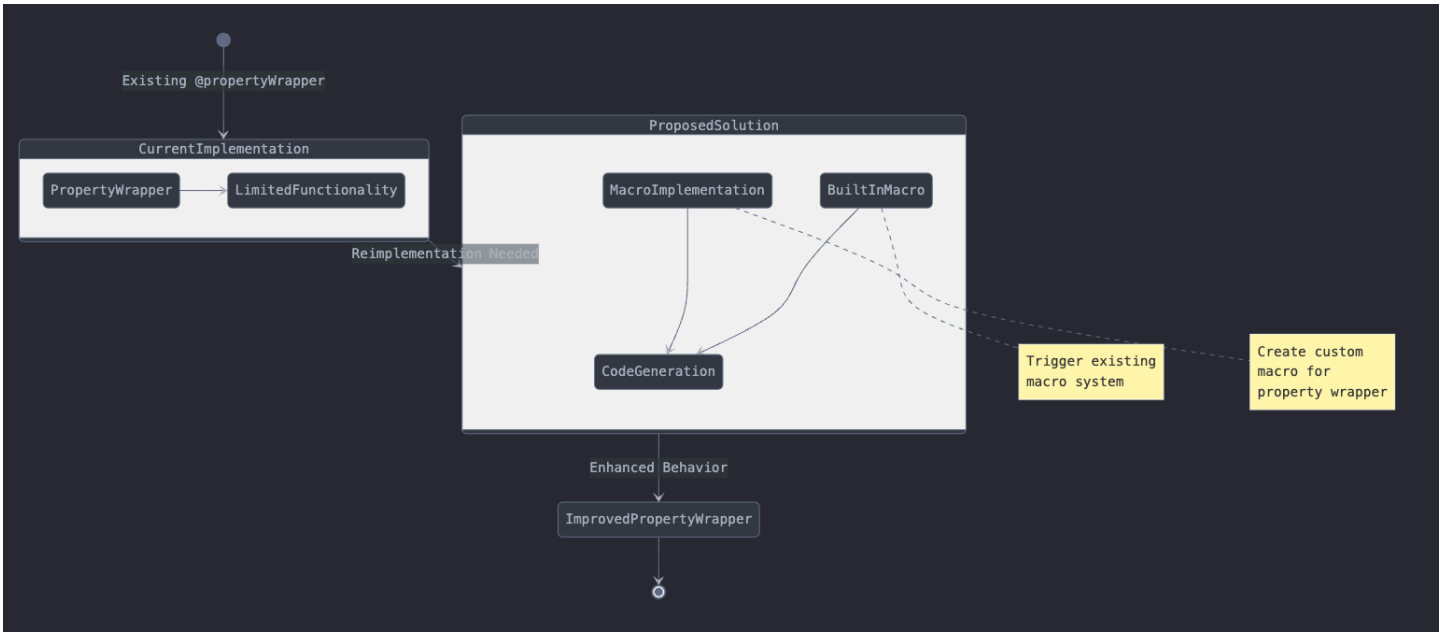
Solution: Create a comprehensive capability audit table comparing property wrapper features to macro capabilities, and extend the macro system where needed.



2. Handling the @propertyWrapper Attribute:

The @propertyWrapper attribute needs to be reimplemented to work with macros.

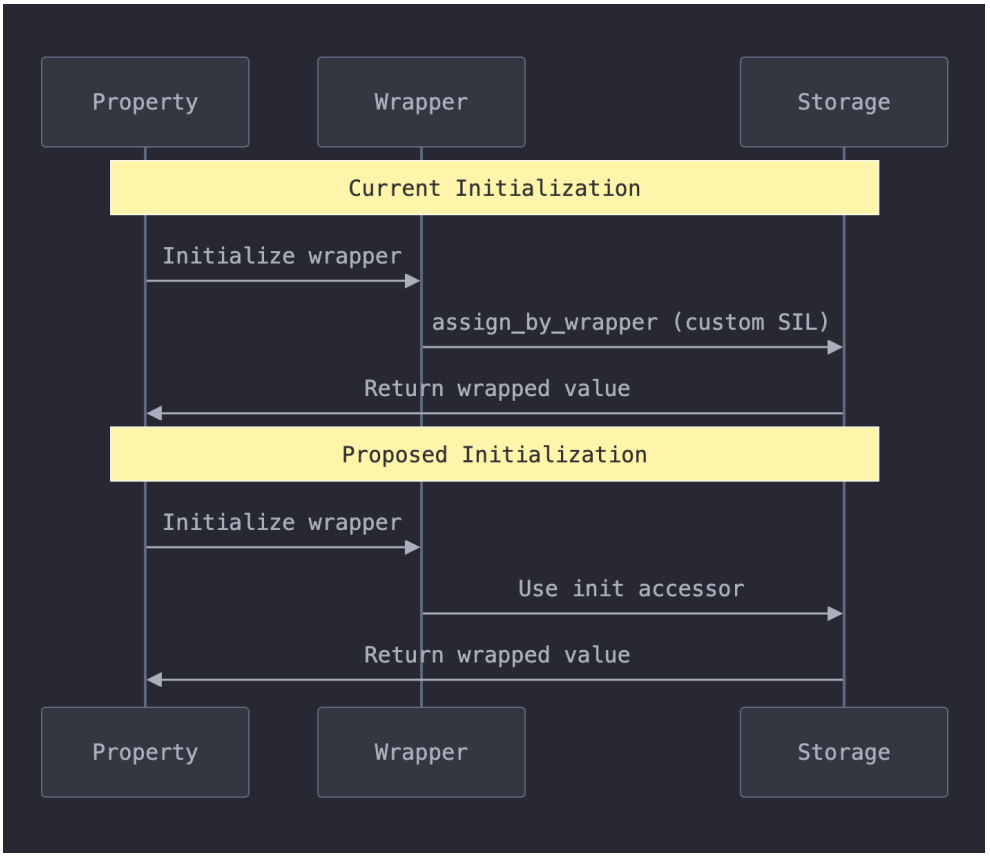
Solution: Convert @propertyWrapper to either trigger a built-in macro or become a macro itself that handles code generation for property wrapper behaviour.



3. Initialization with Init Accessors

Replace the custom assign_by_wrapper SIL instruction with the standard init accessor mechanism.

Solution: Develop a migration path from the custom SIL instruction to init accessors that maintain identical behaviour.



4. Type Inference Challenges

Challenge: Property wrappers participate in Swift's bidirectional type inference, where types flow from wrapper to wrapped value and vice versa. Maintaining this behavior with macros requires careful integration with the type checker.

Solution:

- Create a specialized type inference context for property wrapper macros
- Implement placeholder type variables that get resolved during type checking
- Develop constraint propagation mechanisms that maintain bidirectional type flow
- Integrate with Swift's constraint solver to handle complex type relationships
- Add specialized diagnostic handling for type inference failures
- Create type inference tests for all property wrapper type scenarios
- Implement caching for type inference results to maintain performance
- Design clear interfaces between macro expansion and type checking phases

5. Preserving Projected Value Semantics

Challenge: Property wrappers provide projected values (accessed with $) that have special syntax and semantics, which needs to be maintained in the macro implementation.

Solution:

- Implement specialized macro expansion for projected value access
- Create a consistent naming scheme for generated properties
- Handle visibility and access control for projected values
- Support inheritance of projected value types
- Implement diagnostic handling for projected value access errors
- Create test suite for projected value semantics
- Support composition of projected values in nested property wrappers
- Integrate with Swift's member lookup system for projected value resolution

6. Function Parameter Property Wrappers

Challenge: Property wrappers can be applied to function parameters, which introduces additional complexity for parameter passing, initialization, and call-site transformation

Solution:

- Extend macros to handle function parameter contexts
- Implement call-site transformations for property wrapper parameters
- Handle argument conversion and validation at call sites
- Support default values for wrapped parameters

- Handle variadic parameters with property wrappers
- Create specialized diagnostic handling for parameter wrappers
- Develop test cases focusing on parameter property wrappers
- Support composition of multiple parameter wrappers

7. Performance Considerations

Challenge: Macro-based implementations may introduce performance overhead in both compile-time and run-time compared to the specialized compiler implementation.

Solution:

- Develop comprehensive performance benchmarks for compilation time
- Create run-time performance tests for property wrapper access patterns
- Implement optimization passes specifically for generated property wrapper code
- Add caching mechanisms for macro expansion results
- Reduce AST size through efficient code generation
- Implement specialized SIL optimizations for common property wrapper patterns
- Compare performance metrics before and after implementation
- Create performance regression tests to prevent future slowdowns

---

Current Implementation Analysis:

A deep dive into the current property wrapper implementation reveals specialized handling at multiple compiler stages:

1. **Parsing Phase**
    - Recognition of the @propertyWrapper attribute
    - Special parsing rules for property wrappers applied to properties
    - Special handling for property wrappers applied to function parameters

2. **AST Building Phase**
    - Creating specialized AST nodes for property wrappers
    - Handling for synthesized storage properties and projection properties

3. **Semantic Analysis**
    - Special type-checking rules for property wrapper initialization
    - Validation of property wrapper types
    - Resolution of synthesized storage and accessor methods

4. **SIL Generation**
    - Custom assign_by_wrapper SIL instruction for initialization
    - Special handling for wrapped property access

5. **IRGen Phase**
    - Special codegen for property wrapper access patterns

Each of these steps requires **custom compiler logic**, making Swift's compiler more complex. The goal of this project is to **remove these special cases** and handle property wrappers using **macros and init accessors.**

Proposed Implementation

1. Implementation plan (swift level)

   1.1 Converting @propertyWrapper to a Declaration Macro

```swift
//  Created by Arihant Marwaha
//
import Foundation
// Declaration macro for property wrapper types
@attached(member)
@attached(conformance)
@attached(peer)
public macro propertyWrapper() = #externalMacro(
    module: "SwiftBuiltinMacros",
    type: "PropertyWrapperMacro"
)
```

The declaration macro will perform these operations:

- Validate that the type has a wrappedValue property
- Register the type in the compiler's type system as a valid property wrapper
- Generate necessary accessor methods
- Apply additional attached macros to handle property expansions

Validation Logic

```swift
//  Created by Arihant Marwaha
//
import Foundation
// Inside PropertyWrapperMacro implementation
func validatePropertyWrapperType(_ type: some TypeSyntaxProtocol) throws {
    // Verify the type has a wrappedValue property
    guard let wrappedValueProperty = type.findProperty(named: "wrappedValue") else {
        throw MacroError.missingWrappedValue
    }

    // Verify wrappedValue has appropriate access level
    guard wrappedValueProperty.hasAccessibleGetterAndSetter() else {
        throw MacroError.inaccessibleWrappedValue
    }
}
```

1.2 Implementing Accessor Macros for Property Wrapper Expansion:

An accessor macro will handle the expansion of properties decorated with property wrappers

```swift
//  Created by Arihant Marwaha
import Foundation
public struct PropertyWrapperAccessorMacro: AccessorMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingAccessorsOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AccessorDeclSyntax] {
        // Extract property wrapper type and arguments
        guard let propertyDecl = declaration.as(VariableDeclSyntax.self) else {
            throw MacroError.invalidDeclaration
        }

        // Generate getter accessor
        let getterBody = """
        return _$storage.wrappedValue
        """

        // Generate setter accessor
        let setterBody = """
        _$storage.wrappedValue = newValue
        """

        // Generate init accessor
        let initBody = """
        _$storage = PropertyWrapperType(wrappedValue: initialValue)
        """

        // Return all accessors
        return [
            AccessorDeclSyntax(accessorSpecifier: "get", body: getterBody),
            AccessorDeclSyntax(accessorSpecifier: "set", body: setterBody),
            AccessorDeclSyntax(accessorSpecifier: "init", body: initBody)
        ]
    }
```

1.3 Implementing Member Macro for Backing Storage :Handle any additional members needed for the wrapper's functionality

```swift
//  Created by Arihant Marwaha
public struct PropertyWrapperMemberMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {
        guard let propertyDecl = declaration.as(VariableDeclSyntax.self),
              let binding = propertyDecl.bindings.first,
              let identifier =
                  binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let wrapperType = extractWrapperType(from: node) else {
            throw MacroError.invalidPropertyWrapperUsage
        }

        // Generate backing storage property
        let storageName = "_$\(identifier)"
        let storageDecl = """
        private var \(storageName): \(wrapperType)
        """

        // Generate projected value property if needed
        var declarations: [DeclSyntax] = [DeclSyntax(stringLiteral: storageDecl)]

        if hasProjectedValue(wrapperType) {
            let projectedValueDecl = """
            var $\(identifier): \(wrapperType.projectedValueType) {
                return _$\(identifier).projectedValue
            }
            """
            declarations.append(DeclSyntax(stringLiteral: projectedValueDecl))
        }

        return declarations
    }
```

1.4 Replacing `assign_by_wrapper` SIL Instruction with Init Accessors:

Currently, property wrapper initialization uses a special SIL (Swift Intermediate Language) instruction called `assign_by_wrapper`. The proposal suggests replacing this with standard init accessors.

```
//  Created by Arihant Marwaha

//current implementation
// Conceptual SIL representation
assign_by_wrapper %wrappedValue, to %storage : $PropertyWrapper

//proposed new implementation
// Generated init accessor in SIL (conceptual)
func init(initialValue: T) {
    self._$storage = PropertyWrapper(wrappedValue: initialValue)
}
```

The compiler will generate code that initializes the property using the init accessor.This would eliminate the need for special SIL instructions while maintaining the same behavior

```
// When a user writes:
struct User {
    @PropertyWrapper
    var name: String

    init(name: String) {
        self.name = name  // This will use the init accessor
    }
}

// The expanded code will use:
extension User {
    init(name initialValue: String) {
        self._$name = PropertyWrapper(wrappedValue: initialValue)
    }
}
```

1.5 Handling Function Parameter Property Wrappers:

Property wrappers can also be applied to function parameters, which requires special transformation at the call site.

When calling this function, the compiler currently transforms the parameter to use the wrapper. The proposal suggests implementing this transformation using macros instead of special compiler code

```
public struct FunctionParameterPropertyWrapperMacro: FunctionParameterMacro {
    public static func expansion(
        of attribute: AttributeSyntax,
        attachedTo parameter: FunctionParameterSyntax,
        providingParametersIn function: FunctionDeclSyntax,
        in context: some MacroExpansionContext
    ) throws -> [FunctionParameterSyntax] {
        guard let wrapperType = extractWrapperType(from: attribute),
              let paramName = parameter.firstName?.text else {
            throw MacroError.invalidParameterWrapper
        }

        // Transform parameter to use the wrapped value
        let transformedParameter = parameter.with(
            type: wrapperType.wrappedValueType,
            defaultArgument: nil
        )

        return [transformedParameter]
    }
```

```
    public static func expansion(
        of attribute: AttributeSyntax,
        attachedTo parameter: FunctionParameterSyntax,
        providingStatementsIn function: FunctionDeclSyntax,
        in context: some MacroExpansionContext
    ) throws -> [CodeBlockItemSyntax] {
        guard let wrapperType = extractWrapperType(from: attribute),
              let paramName = parameter.firstName?.text else {
            throw MacroError.invalidParameterWrapper
        }

        // Generate wrapper initialization code
        let wrapperInit = """
        let _$\(paramName) = \(wrapperType)(wrappedValue: \(paramName))
        """

        // Replace parameter references with wrapper access
        let parameterTransform = """
        // Access wrapped value through the wrapper
        let \(paramName) = _$\(paramName).wrappedValue
        """

        return [
            CodeBlockItemSyntax(stringLiteral: wrapperInit),
            CodeBlockItemSyntax(stringLiteral: parameterTransform)
        ]
    }
}
```

1.6 Supporting Projected Values

Property wrappers often include "projected values" which require additional handling and this proposed way could be helpful in supporting the above arguments. By generating the appropriate properties and accessors for the projected values.

```swift
//  Created by Arihant Marwaha
public struct ProjectedValueMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {
        guard let propertyDecl = declaration.as(VariableDeclSyntax.self),
              let binding = propertyDecl.bindings.first,
              let identifier =
                  binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let wrapperType = extractWrapperType(from: node),
              hasProjectedValue(wrapperType) else {
            return []
        }

        // Generate projected value accessor
        let projectedValueDecl = """
        var $\(identifier): \(wrapperType.projectedValueType) {
            return _$\(identifier).projectedValue
        }
        """

        return [DeclSyntax(stringLiteral: projectedValueDecl)]
    }
}
```

2. Implementation plan (compiler level):

Reimplementing property wrappers with macros requires changes across multiple compiler subsystems:

1. Parser and AST generation

1.1 Transitioning from Attribute to Macro

Current implementation in `lib/Parse/ParseDecl.cpp`:

```
bool Parser::parseAttributePrefix(DeclAttributes &Attributes,
    bool inContext,
    bool checkAttributeList) {
// ...
if (Tok.getText() == "propertyWrapper") {
// Process as special attribute with custom parsing rules
auto *AttrInfo = parsePropertyWrapperAttribute();
Attributes.add(AttrInfo);
return true;
}
// ...
}

PropertyWrapperAttr *Parser::parsePropertyWrapperAttribute() {
// Complex custom parsing logic
// ...
return new (Context) PropertyWrapperAttr(/*implicit*/false);
}
```

Proposed changes to `lib/Parse/ParseDecl.cpp`:

```
// New implementation
bool Parser::parseAttributePrefix(DeclAttributes &Attributes,
    bool inContext,
    bool checkAttributeList) {
// ...
if (Tok.getText() == "propertyWrapper") {
// Register as a macro instead of special attribute
auto *AttrInfo = parseBuiltinMacroAttribute("propertyWrapper");

// Record for macro expansion
auto *MacroRef = MacroRegistry::get().lookupMacro("propertyWrapper");
Attributes.add(new (Context) MacroApplicationAttr(MacroRef));
return true;
}
```

1.2 AST Node Changes

Remove custom AST nodes in `include/swift/AST/Attr.h`

`[To be removed]`

```
class PropertyWrapperAttr : public DeclAttribute {
    // Property wrapper specific attributes
    // ...
  public:
    PropertyWrapperAttr(bool implicit)
      : DeclAttribute(DAK_PropertyWrapper, SourceLoc(), SourceRange(),
                      /*Implicit=*/implicit) {}

    static bool classof(const DeclAttribute *DA) {
      return DA->getKind() == DAK_PropertyWrapper;
    }
  };
```

Replace with macro-based implementation in `include/swift/AST/MacroDef.h`

```
//new implementation
class PropertyWrapperMacro : public DeclarationMacro {
    public:
      PropertyWrapperMacro() : DeclarationMacro("propertyWrapper") {}

      MacroExpansionResult expand(MacroExpansionContext &context,
                                  const MacroApplicationExpr *application,
                                  const DeclContext *declContext) override;

      bool validateTargetNode(ASTContext &ctx,
                              const Decl *target,
                              DiagnosticEngine &diags) const override;
    };
```

1.3 Registration in Compiler : Add to lib/AST/MacroRegistry.cpp

```
3   void MacroRegistry::registerBuiltinMacros() {
4       // Register other built-in macros
5       // ...
6
7       // Register property wrapper macro
8       registerMacro(std::make_unique<PropertyWrapperMacro>());
9   }
```

2. Type checking and semantic analysis

2.1 Type Checking for Property Wrappers

Current implementation in `lib/Sema/TypeCheckAttr.cpp` [To be removed]

```
void TypeChecker::checkPropertyWrapperAttributes(VarDecl *var) {
    // Complex validation for property wrappers
    auto *attr = var->getAttrs().getAttribute<PropertyWrapperAttr>();
    if (!attr)
      return;

    // Perform validation and setup
    if (!validatePropertyWrapperType(var, attr))
      return;

    // Set up backing storage and accessors
    setupPropertyWrapperStorage(var, attr);
  }
```

Replace with macro expansion handling in `lib/Sema/TypeCheckMacros.cpp`:

```
void MacroExpander::expandPropertyWrapperMacro(VarDecl *var,
    MacroApplicationExpr *macroApp) {
// Set up macro expansion context
MacroExpansionContext context(Ctx, var->getDeclContext());

// Expand the macro to get transformed declarations
auto expansionResult = PropertyWrapperMacro().expand(context, macroApp, var->getDeclContext());

if (expansionResult.isError())
return;

// Register expanded declarations in the AST
for (auto *newDecl : expansionResult.getDeclarations()) {
var->getDeclContext()->addMember(newDecl);
}

// Register expanded accessors
for (auto *accessor : expansionResult.getAccessors()) {
var->addAccessor(accessor);
}
}
```

2.2 Validate Property Wrapper Types :

In `lib/Sema/PropertyWrapperTypeChecker.cpp [To be reomoved]`

```
bool TypeChecker::validatePropertyWrapperType(VarDecl *var, PropertyWrapperAttr *attr) {
    // Validate the property wrapper has wrappedValue
    // Check init requirements
    // Verify access control
    // ...
  }
```

New implementation in lib/Sema/MacroExpansion.cpp:

```
bool PropertyWrapperMacro::validateTargetNode(ASTContext &ctx,
    const Decl *target,
    DiagnosticEngine &diags) const {
// Must be a nominal type with a wrappedValue property
auto *nominalDecl = dyn_cast<NominalTypeDecl>(target);
if (!nominalDecl) {
diags.diagnose(target->getLoc(),
diag::property_wrapper_not_nominal_type);
return false;
}

// Look up wrappedValue property
auto wrappedValueLookup = nominalDecl->lookupDirect(ctx.Id_wrappedValue);
auto wrappedValueProperty = findProperty(wrappedValueLookup);

if (!wrappedValueProperty) {
diags.diagnose(nominalDecl->getLoc(),
diag::property_wrapper_no_wrapped_value);
return false;
}

// Additional validation
// ...

return true;
}
```

3. SIL generation

3.1 Remove Custom SIL Instructions :

In `include/swift/SIL/SILInstruction.h`: [To be removed]

```
class AssignByWrapperInst : public SILInstruction {
    private:
      SILValue WrappedValue;
      SILValue Storage;

    public:
      static AssignByWrapperInst *create(/*...*/);
      // ...
    };
```

3.2 Replace with Standard Init Accessors

In `lib/SILGen/SILGenApply.cpp` [To be removed]

```
void SILGenFunction::emitAssignByWrapper(SILLocation loc,
    SILValue wrappedValue,
    SILValue storage) {
// Create custom SIL instruction
auto *inst = AssignByWrapperInst::create(loc, wrappedValue, storage,
    SGM.M);
B.insert(inst);
}
```

Replaced with

```
4  void SILGenFunction::emitPropertyWrapperInit(SILLocation loc,
5      SILValue initialValue,
6      AbstractStorageDecl *storage) {
7    // Lookup the init accessor
8    SILDeclRef initRef(storage->getOpaqueAccessor(AccessorKind::Init)
9    SILDeclRef::Kind::Accessor);
10
11   // Generate standard accessor call
12   auto accessorFn = SGM.emitDeclRef(initRef, loc);
13
14   // Prepare arguments
15   SILValue selfValue = emitStorageAddressForProperty(storage);
16
17   // Generate standard function call
18   B.createApply(loc, accessorFn, {selfValue, initialValue}, false);
19   }
```

3.3 SIL Lowering for Property Initialization

In `lib/SILGen/SILGenDecl.cpp`: [To be modified]

```
void SILGenFunction::emitPropertyWrapperInit(VarDecl *var, Expr *initExpr) {
    // Get storage reference and initial value
    SILValue storage = emitStorageAddressForProperty(var);
    SILValue initValue = emitRValueAsOrig(initExpr);

    // Use custom instruction
    emitAssignByWrapper(var->getLoc(), initValue, storage);
  }
```

Modified Code :

```
void SILGenFunction::emitPropertyWrapperInit(VarDecl *var, Expr *initExpr) {
    // Get initial value
    SILValue initValue = emitRValueAsOrig(initExpr);

    // Use standard accessor call instead of custom instruction
    emitPropertyWrapperInitAccessor(var->getLoc(), initValue, var);
  }
```

4. Macro Expansion Implementation

4.1 Core Property Wrapper Macro Implementation

The core property wrapper macro implementation needs to handle several aspects of the current property wrapper functionality:

The key components should include:

Type Validation: Ensure the annotated type meets all requirements for a property wrapper

Storage Generation: Create backing storage for wrapped properties

Accessor Generation: Generate getter/setter methods for property access

Projected Value Handling: Support for projected values with $ prefix

Initialization Logic: Handle initialization patterns for wrapped properties

```
MacroExpansionResult
PropertyWrapperMacro::expand(MacroExpansionContext &context,
                             const MacroApplicationExpr *application,
                             const DeclContext *declContext) {
  ASTContext &ctx = context.getASTContext();
  DiagnosticEngine &diags = ctx.Diags;

  // Validate the target is a var declaration
  auto *var = dyn_cast<VarDecl>(application->getTarget());
  if (!var) {
    diags.diagnose(application->getLoc(),
                   diag::property_wrapper_not_applied_to_var);
    return MacroExpansionResult::error();
  }
```

```
  // Extract property wrapper type and arguments
  Type wrapperType = application->getGenericArgs()[0];
  NominalTypeDecl *wrapperTypeDecl = wrapperType->getAnyNominal();

  // Generate backing storage declaration
  auto *storageVar = createBackingStorageVar(ctx, var, wrapperType);

  // Generate accessors
  auto *getterDecl = createGetter(ctx, var, storageVar);
  auto *setterDecl = createSetter(ctx, var, storageVar);
  auto *initDecl = createInitAccessor(ctx, var, storageVar);

  // Generate projected value property if needed
  VarDecl *projectedValueVar = nullptr;
  if (hasProjectedValue(wrapperTypeDecl)) {
    projectedValueVar = createProjectedValueVar(ctx, var, storageVar);
```

The build result would look something like this :

```
// Build result
MacroExpansionResult result;
result.addDeclaration(storageVar);
if (projectedValueVar)
  result.addDeclaration(projectedValueVar);

result.addAccessor(getterDecl);
result.addAccessor(setterDecl);
result.addAccessor(initDecl);

return result;
}
```

To address the type *inference challenges*, a recommendation is to implement a **two-phase expansion** approach where the first phase would do structural transformation only and then the second phase would apply type information to the structural transformation.

The proposed implementation could look something like this in the given segments:

```
public static func phaseOneExpansion(
  of node: AttributeSyntax,
  attachedTo declaration: some DeclSyntaxProtocol,
  in context: some MacroExpansionContext
) throws -> PropertyWrapperStructure {
  // First phase: structural transformation only
  // Return an intermediate representation with type placeholders
}

public static func phaseTwoExpansion(
  of structure: PropertyWrapperStructure,
  withTypeInfo typeInfo: TypeInferenceContext,
  in context: some MacroExpansionContext
) throws -> [DeclSyntax] {
  // Second phase: apply type information to the structural transformation
  // Complete the expansion with concrete types
}
```

4.2 Helper Functions for Macro Implementation:

Creating several helper functions will make the implementation more modular and maintainable.

Swift macro implementation:

1. Validates that the type can be used as a property wrapper

```
func validatePropertyWrapperType(
  declaration: DeclGroupSyntax,
  context: some MacroExpansionContext
) throws -> Bool {
  // Check for wrappedValue property
  // Validate initializers
  // Check for required methods
}
```

2. Generates the backing storage property

```
func generateBackingStorage(
  forProperty property: VariableDeclSyntax,
  withWrapperType wrapperType: TypeSyntax,
  initExpr: ExprSyntax?
) -> VariableDeclSyntax {
  // Create _$storage variable with appropriate attributes
}
```

3. Generates property accessor methods

```
func generateAccessors(
  forProperty property: VariableDeclSyntax,
  storageRef: ExprSyntax
) -> [AccessorDeclSyntax] {
  // Create getter that accesses wrappedValue
  // Create setter that updates wrappedValue
}
```

4. Handles projected value synthesis

```
func generateProjectedValue(
  forProperty property: VariableDeclSyntax,
  wrapperType: TypeSyntax
) -> [DeclSyntax] {
  // Create projection property with $ prefix if needed
}
```

5. Type inference helper

```
func inferPropertyType(
  wrapperType: TypeSyntax,
  valueType: TypeSyntax?,
  context: TypeInferenceContext
) -> TypeSyntax {
  // Apply type inference rules to determine concrete types
}
```

6. Init accessor generation

```
func generateInitAccessor(
  forProperty property: VariableDeclSyntax,
  wrapperType: TypeSyntax,
  valueExpr: ExprSyntax?
) -> AccessorDeclSyntax {
  // Generate init accessor that replaces assign_by_wrapper
}
```

For the type inference challenges, I propose creating a specialized TypeInferenceContext that interacts with the compiler's type system:

```
struct TypeInferenceContext {
  // References to compiler type system
  let typeChecker: TypeCheckerRef
  let constraintSystem: ConstraintSystemRef

  // Methods to interact with type inference
  func resolveWrappedType(for wrapperType: TypeSyntax) -> TypeSyntax
  func resolveWrapperType(for valueType: TypeSyntax) -> TypeSyntax
  func createTypePlaceholder() -> TypeSyntax
  func constrainType(_ type: TypeSyntax, to constraint: TypeSyntax)
}
```

To implement this, you'll need to extend the macro system API to provide access to the compiler's type inference system. This would require:

1. Integration points in the compiler pipeline - Define clear interfaces between macro expansion and type checking phases
2. Type placeholder mechanism - Allow macros to create type variables that get resolved during type checking
3. Constraint propagation - Ensure type constraints flow bidirectionally between wrapper and wrapped value

For compiler level implementation I would suggest making the following helper methods :

1. Helper to create backing storage variable:

```
VarDecl *createBackingStorageVar(ASTContext &ctx,VarDecl *originalVar,Type wrapperType) {
// Create mangled name for storage property
Identifier storageName =
ctx.getIdentifier(("_$" + originalVar->getName().str()).str());

// Create storage variable declaration
auto *storageVar = new (ctx) VarDecl(
/*IsStatic=*/originalVar->isStatic(),
VarDecl::Specifier::Var,
/*IsLet=*/false,
originalVar->getLoc(),
storageName,
originalVar->getDeclContext());

// Set type, access level, and other properties
storageVar->setType(wrapperType);
storageVar->setAccess(AccessLevel::Private);

// Create pattern binding for initialization
// ...
return storageVar;
}
```

2. Helper to create getter accessor

```
// Helper to create getter accessor
AccessorDecl *createGetter(ASTContext &ctx,
VarDecl *originalVar,
VarDecl *storageVar) {
// Create getter function
auto *getterDecl = AccessorDecl::create(
ctx,
originalVar->getLoc(),
originalVar->getLoc(),
AccessorKind::Get,
originalVar,
/*async*/false,
/*throws*/false,
/*parameters*/nullptr,
Type(),
originalVar->getDeclContext());
// Create function body with "return _$storage.wrappedValue"
// ...
return getterDecl;
}
```

3. Helper to create setter accessor

```
// Helper to create setter accessor
AccessorDecl *createSetter(ASTContext &ctx,
VarDecl *originalVar,
VarDecl *storageVar) {
// Create setter with "self._$storage.wrappedValue = newValue"
// ...
}
```

4. Helper to create init accessor

```
// Helper to create init accessor
AccessorDecl *createInitAccessor(ASTContext &ctx,
    VarDecl *originalVar,
    VarDecl *storageVar) {
// Parameter for initialValue
auto *paramDecl = new (ctx) ParamDecl(
/*specifier*/ParameterTypeFlags::Default,
SourceLoc(),
ctx.getIdentifier("initialValue"),
originalVar->getLoc(),
ctx.getIdentifier("initialValue"),
originalVar->getDeclContext());
```

```
paramDecl->setType(originalVar->getType());
ParameterList *params = ParameterList::create(ctx, {paramDecl});
// Create init accessor
auto *initDecl = AccessorDecl::create(
ctx,
originalVar->getLoc(),
originalVar->getLoc(),
AccessorKind::Init,
originalVar,
/*async*/false,
/*throws*/false,
params,
Type(),
originalVar->getDeclContext());
// Create body with "self._$storage = Wrapper(wrappedValue: initialValue)"
// ...
return initDecl;
}
```

5. Diagnostic handling

5.1 Update Diagnostic Definitions

5.2 Update Error Handling

6. Migration Strategy

6.1 Incremental Feature Flag:Add compiler flag in include/swift/Basic/LangOptions.h

6.2 AST Compatibility Layer :Ensure backward compatibility in lib/AST/ASTWalker.cpp

6.3 Dual-Path Implementation : Implement a compatibility layer during transition

7. Testing Infrastructure

7.1 Unit Tests for Property Wrapper Macro

7.2 SIL Tests to Verify Generated Code

7.3 Migration Tests

7.4 Performance Comparison Tests

8. Binary Compatibility Considerations

8.1 ABI Stability [When migrating from attributes to macros, we need to ensure ABI stability for existing property wrapper code]

8.2 Module Interface Changes

9. Runtime Support Changes

9.1 Dynamic Property Lookup

9.2 Serialization Format

10. IDE Support Updates

10.1 Code Completion

10.2 Refactoring Engine

10.3 SourceKit LSP Support

11. Performance Optimizations

    12.1 Reducing Expanded AST Size

    12.2 Caching Expansion Results

    12.3 SIL-Level Optimizations

    12.4 Performance Benchmarking and Validation

Expected Benefits from the Completion of this project :

Reduced compiler complexity

- Removal of special-case code throughout the compiler
- More consistent handling of language features

Improved maintainability

- Less specialized code to maintain
- Fewer special cases to understand and document

Better testability

- Property wrapper behaviour can be tested using standard macro testing tools
- Easier to validate behaviour in isolation

Fixed corner cases

- The current implementation has known corner cases and limitations
- A macro-based implementation can address these issues

Simplified future extensions

- New property wrapper features can be added by extending macros
- Reduced need for compiler changes to add new capabilities

Testing Stratergy

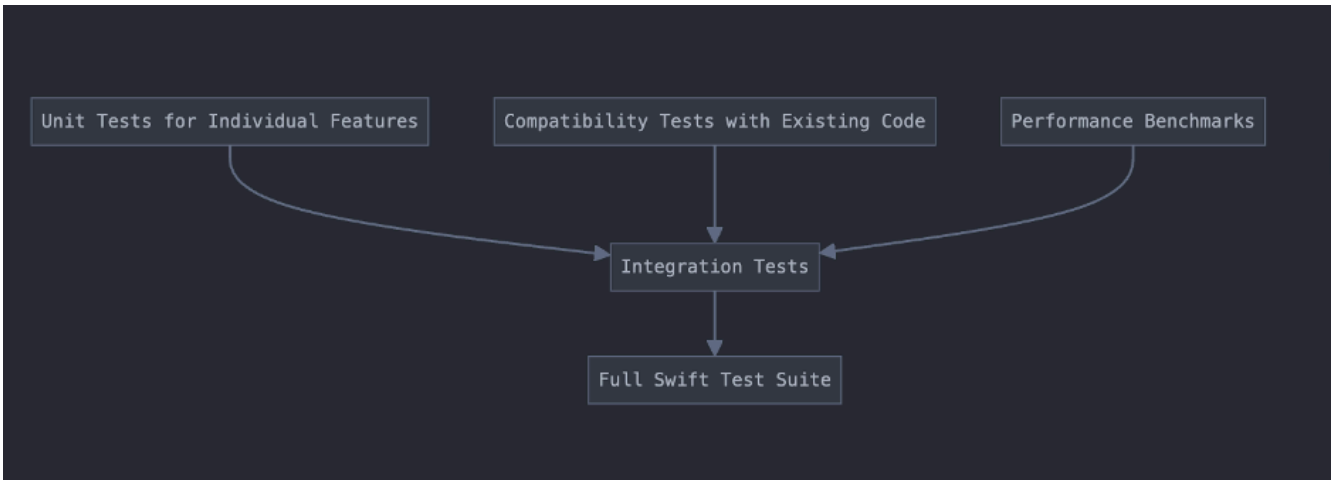
Unit tests for each property wrapper feature

Migration tests to verify identical behaviour before and after the change

Performance tests to measure any impact on compile time or runtime performance

Edge case tests for complex property wrapper usage

Integration tests with the rest of the Swift ecosystem



Example Test Cases :

```swift
// Basic property wrapper test
func testBasicPropertyWrapper() {
    struct TestWrapper<T> {
        var wrappedValue: T
    }

    struct TestStruct {
        @TestWrapper var value: Int = 42
    }

    var test = TestStruct()
    XCTAssertEqual(test.value, 42)
    test.value = 100
    XCTAssertEqual(test.value, 100)
    XCTAssertEqual(test._$value.wrappedValue, 100)
}
```

```swift
// Projected value test
func testProjectedValue() {
    struct TestWrapper<T> {
        var wrappedValue: T
        var projectedValue: Binding<T> {
            Binding(get: { self.wrappedValue },
                    set: { self.wrappedValue = $0 })
        }
    }

    struct TestStruct {
        @TestWrapper var value: Int = 42
    }

    var test = TestStruct()
    test.$value.wrappedValue = 100
    XCTAssertEqual(test.value, 100)
}
```

Deliverables

Community Bonding Period

- Engage in discussions with mentors and the broader Swift community to refine the implementation strategy.
- Identify and document necessary improvements to the macro system that will support property wrapper expansion.
- Develop a well-structured roadmap outlining incremental steps for implementing macro-based property wrappers while ensuring minimal disruption to the existing system.

First Evaluation

- Implement a basic mechanism to expand property wrappers using macros, laying the groundwork for more advanced functionality.
- Introduce macro-based storage properties to replace traditional compiler-generated storage.
- Write and execute unit tests for fundamental use cases, verifying the correctness of simple property wrappers under macro expansion.

Second Evaluation

- Implement integration with the **init** accessor to support property wrappers that rely on initialization behavior.
- Extend macros to support property wrappers used in function parameters, enabling their usage in method signatures.
- Develop a comprehensive set of test cases to validate function parameter applications of macro-based property wrappers.

Final Evaluation

- Ensure full backward compatibility with existing property wrappers, allowing seamless transition for Swift developers.
- Conduct performance testing and optimizations to ensure macro-based property wrappers do not introduce significant overhead.
- Submit pull requests (PRs) for code review, address feedback from the Swift community, and contribute to official documentation to aid adoption and understanding.

Detailed Implementation Phases and Milestones

Phase 1: Analysis and Foundation (Weeks 1-4)

Week 1: Compiler Architecture Analysis

- Task 1.1: Map all property wrapper touchpoints in the Swift compiler
    - Deliverable: Comprehensive document identifying all property wrapper specific code
    - Steps:
        1. Identify all occurrences of property wrapper handling in parser
        2. Document property wrapper AST nodes and their creation
        3. Map property wrapper type checking and validation code
        4. Identify all SIL generation code for property wrappers
        5. Document code generation patterns for property wrappers
- Task 1.2: Document the runtime behavior of property wrappers
    - Deliverable: Behavioral specification document
    - Steps:
        1. Create test cases for each property wrapper feature
        2. Document the exact runtime behavior with memory layout diagrams
        3. Create assembly-level mappings of property wrapper operations

Week 2: Macro System Gap Analysis

- Task 2.1: Analyze current macro system capabilities
    - Deliverable: Capability comparison matrix
    - Steps:
        1. Document all current macro types and their abilities
        2. Map property wrapper features to existing macro capabilities
        3. Identify capability gaps that need to be addressed
- Task 2.2: Prototype macro system extensions
    - Deliverable: Proposal for macro system extensions
    - Steps:
        1. Design new macro types or extensions needed
        2. Create minimal implementation prototypes
        3. Document API changes required

Week 3: Init Accessor Analysis

- Task 3.1: Analyze assign_by_wrapper SIL instruction
    - Deliverable: Technical specification document
    - Steps:
        1. Document exact behavior of assign_by_wrapper
        2. Create control flow graphs for current implementation
        3. Identify edge cases and special handling

- Task 3.2: Map to init accessor patterns
    - Deliverable: Init accessor implementation specification
    - Steps:
        1. Design init accessor patterns for property wrappers
        2. Create mapping from assign_by_wrapper to init accessors
        3. Identify any semantic differences that need addressing

Week 4: Architecture Design

- Task 4.1: Design macro-based property wrapper architecture
    - Deliverable: Architecture specification document
    - Steps:
        1. Create component diagram for new implementation
        2. Define interfaces between components
        3. Document code generation patterns
- Task 4.2: Create proof-of-concept implementation
    - Deliverable: Minimal working prototype
    - Steps:
        1. Implement basic property wrapper macro
        2. Test with simple use cases
        3. Validate against current implementation

Phase 2: Core Implementation (Weeks 5-10)

Week 5-6: Macro System Extensions

- Task 5.1: Implement property wrapper macro type
    - Deliverable: Property wrapper macro type implementation
    - Steps:
        1. Create new macro type definition
        2. Implement expansion logic
        3. Add diagnostic handling
- Task 5.2: Add function parameter support
    - Deliverable: Function parameter macro capabilities
    - Steps:
        1. Extend macro system to handle function parameters
        2. Implement code injection at function boundaries
        3. Add support for parameter access and transformation

Week 7-8: Init Accessor Integration

- Task 7.1: Implement init accessor generation
    - Deliverable: Init accessor code generation
    - Steps:
        1. Create code generation patterns for init accessors

2. Implement initialization logic
3. Handle different initialization scenarios

- Task 7.2: Replace assign_by_wrapper with init accessors
  - Deliverable: SIL generation using init accessors
  - Steps:
    1. Create new SIL generation paths using init accessors
    2. Implement fallback paths for backward compatibility
    3. Add validation and verification

Week 9-10: AST Transformation

- Task 9.1: Implement property wrapper AST transformation
  - Deliverable: AST transformation implementation
  - Steps:
    1. Create AST visitor for property wrapper identification
    2. Implement transformation logic
    3. Add validation and error handling
- Task 9.2: Integrate with type checking
  - Deliverable: Type checking integration
  - Steps:
    1. Implement type checking for transformed AST
    2. Add constraint solving for property wrapper types
    3. Implement diagnostic handling

Phase 3: Testing and Refinement (Weeks 11-16)

Week 11-12: Basic Testing

- Task 11.1: Implement unit tests
  - Deliverable: Unit test suite
  - Steps:
    1. Create tests for core functionality
    2. Implement regression tests for known issues
    3. Add edge case tests
- Task 11.2: Implement functional tests
  - Deliverable: Functional test suite
  - Steps:
    1. Create tests for end-to-end functionality
    2. Implement compatibility tests
    3. Add performance benchmarks

Week 13-14: Comprehensive Testing

- Task 13.1: Implement corner case tests
  - Deliverable: Corner case test suite

    - Steps:
        1. Identify and document corner cases
        2. Implement tests for each corner case
        3. Validate results against current implementation
- Task 13.2: Ecosystem integration tests
    - Deliverable: Ecosystem test suite
    - Steps:
        1. Test with SwiftUI and other standard library users
        2. Test with third-party libraries
        3. Validate against real-world code

Week 15-16: Performance Optimization

- Task 15.1: Measure and optimize compilation performance
    - Deliverable: Performance optimization report
    - Steps:
        1. Create benchmarks for compilation time and memory use
        2. Identify bottlenecks
        3. Implement optimizations
- Task 15.2: Measure and optimize runtime performance
    - Deliverable: Runtime optimization report
    - Steps:
        1. Create benchmarks for runtime performance
        2. Compare with current implementation
        3. Implement optimizations

Phase 4: Finalization and Documentation (Weeks 17-20)

Week 17-18: Final Integration

- Task 17.1: Clean up and remove legacy code
    - Deliverable: Clean codebase
    - Steps:
        1. Identify all legacy property wrapper code
        2. Create removal plan with fallbacks
        3. Implement clean-up
- Task 17.2: Final validation
    - Deliverable: Validation report
    - Steps:
        1. Run full Swift test suite
        2. Validate against real-world code
        3. Address any remaining issues

Week 19-20: Documentation and Knowledge Transfer

- Task 19.1: Create technical documentation
    - Deliverable: Technical documentation
    - Steps:
        1. Document architecture and implementation
        2. Create developer guides
        3. Document API changes and evolution
- Task 19.2: Create migration guides
    - Deliverable: Migration documentation
    - Steps:
        1. Document migration paths for edge cases
        2. Create troubleshooting guides
        3. Document performance considerations

Expected Outcomes

Technical Improvements

1. **Simplified Compiler Architecture**: By moving property wrapper implementation from the compiler to the macro system, you'll reduce compiler complexity and maintenance burden.
2. **Improved Performance**:
    - Faster compilation times due to reduced compiler complexity
    - Potential runtime performance improvements through optimized code generation
    - More efficient memory usage patterns
3. **Enhanced Extensibility**: The macro-based approach will make it easier to add new property wrapper features without modifying the compiler.
4. **Better Diagnostics**: Error messages and compiler diagnostics will likely become more precise and developer-friendly.
5. **Reduced Technical Debt**: Replacing the specialized `assign_by_wrapper` SIL instruction with standard init accessors will simplify the SIL layer.

Developer Experience Benefits

1. **Greater Transparency**: Developers will be able to inspect the generated code from property wrappers more easily.
2. **Improved Tooling Support**: IDE features like code completion, refactoring, and documentation will work better with macro-based property wrappers.
3. **Consistency with Other Swift Features**: Property wrappers will align more closely with other Swift language features implemented via macros.
4. **Lower Barrier to Entry**: Creating custom property wrappers may become more accessible to developers familiar with Swift macros.

About Me

I am your friendly neighbourhood but not so regular software engineer who fell in love with the apple architecture the day he saw Steve Jobs take that macbook air out of that envolope. Apple's ability to create functional design items was one part of the fascination but the day I programmed in this low level language called Qbasic at a young age of 10 and nothing has stopped me ever since.

IOS development has always fascinated me because of the seamless design that provides the user with a unique experience of the integrated ecosystem while also maintaining regular functionality. This has led me to study swift in detail and make multiple projects using swift , but now i finally feel like my time has come that i come out of my comfort zone and actually apply those skills into the real world.

I look at programs as not just logic but a way to express myself and my creativity to design efficient and functional products that work. It's been more than a year since I introduced myself to the world of swift and have always wanted to make an impact at the public level. Being a big time gamer , emulation has always fascinated me thus bringing out my love for openSource projects such as PsX2 where the community was able to bring our childhood to our modern devices. Some of my old habits like taking physical notes and making mind maps to organise myself make me an old head but that's how I love to process stuff around me.

For me that's the goal! To create real life impact to the world while sharing my creativity and skills.To be kind, informative and helpful to everyone that needs it.

Why me

I have extensive experience programming in Swift. Recently I also made it to the winners list of **WWDC25 Swift Student challenge 25** for my innovative work. Over the last few days I got the opportunity to introduce myself to the mentors, bond with the community , get instructions to explore the coebase to familiarise myself with what I have to work with. As a result I was able to make notable contributions to solve and propose solutions to certain problems while also understanding certain procedures to make myself comfortable with the open source space.

Here are a few links that show my contributions and also my communication with the community:

1. [Pitch] Compiler-Level Macro Support for Variadic Wrappers
2. [GSoC 2025] Re-implement property wrappers with macros
3. External property wrappers don't play well with variadic parameters #77824
4. Actor-isolated property wrapper leads to compiling code that crashes at runtime #77604
5. Recieved Community Help

I was able to look around the swift-evolution and understand the core functionality of the related topics and declaration within the codebase.

After spending enough time on researching and exploring and making mistakes I feel confident enough to contribute to the project "**Re-implement property wrappers with macros**"