STUNIR Enhancement-to-Emitter Integration

Overview

Phase 1 (Foundation) of the STUNIR Enhancement Integration establishes the integration layer between the enhancement modules (control flow, type system, semantic, memory, optimization) and the target language emitters.

This integration enables intelligent code generation by providing emitters with rich analysis data, allowing them to generate better code with proper control flow structures, type annotations, memory management, and optimizations.

Architecture

```
╔══════════════════════════════════════════════════════════════════╗
║                   ENHANCEMENT PIPELINE                           ║
╚══════════════════════════════════════════════════════════════════╝

  Input IR ──┬──► SemanticAnalyzer ──► SymbolTable
             │          │
             │          ▼
             ├──► ControlFlowAnalyzer ──► CFG (per function)
             │          │
             │          ▼
             ├──► TypeInferenceEngine ──► TypeContext
             │          │
             │          ▼
             ├──► MemoryAnalyzer ──► MemoryContext
             │          │
             │          ▼
             └──► Optimizer (O0-O3) ──► OptimizedIR
                        │
                        ▼
              EnhancementContext (packaged)
                        │
                        ▼
                  TargetEmitter
                        │
                        ▼
               Generated Code (with bodies)
```

Components

EnhancementContext

The `EnhancementContext` class packages all enhancement data into a single object that emitters can consume.

Location: `tools/integration/enhancement_context.py`

Key Features:
- Containers for each enhancement type

- Safe accessors with null-checking
- Serialization/deserialization support
- Validation methods

Data Containers:
- `ControlFlowData`: CFGs, loops, branches, dominators
- `TypeSystemData`: Type mappings, inference results
- `SemanticData`: Symbol tables, call graphs
- `MemoryData`: Memory patterns, safety violations
- `OptimizationData`: Optimization hints, optimized IR

Usage:

```python
from tools.integration import EnhancementContext, create_minimal_context

# Create minimal context for testing
context = create_minimal_context(ir_data, 'rust')

# Access enhancement data safely
cfg = context.get_function_cfg('main')
var_info = context.lookup_variable('x')
memory_strategy = context.get_memory_strategy()

# Serialize/deserialize
json_str = context.to_json()
restored = EnhancementContext.from_json(json_str)
```

EnhancementPipeline

The `EnhancementPipeline` class orchestrates the execution of all enhancement analyses.

Location: `tools/integration/enhancement_pipeline.py`

Key Features:
- Configurable enhancement execution
- Graceful degradation on failures
- Performance tracking
- Comprehensive logging

Usage:

```python
from tools.integration import EnhancementPipeline, PipelineConfig, create_pipeline

# Simple usage
pipeline = create_pipeline('python', 'O2')
context = pipeline.run_all_enhancements(ir_data)

# Custom configuration
config = PipelineConfig(
    enable_control_flow=True,
    enable_type_analysis=True,
    enable_semantic_analysis=True,
    enable_memory_analysis=False,  # Disable memory analysis
    enable_optimization=True,
    optimization_level='O3',
    fail_fast=False  # Continue on failures
)
pipeline = EnhancementPipeline('rust', config)
context = pipeline.run_all_enhancements(ir_data)

# Check results
print(context.get_status_summary())
```

BaseEmitter

The `BaseEmitter` class provides a base class for emitters with enhancement context support.

Location: `tools/emitters/base_emitter.py`

Key Features:
- Optional enhancement context (backward compatible)
- Safe accessors for enhancement data
- File writing utilities
- Manifest generation

Usage:

```python
from tools.emitters.base_emitter import BaseEmitter, EmitterConfig

class PythonEmitter(BaseEmitter):
    TARGET = 'python'
    FILE_EXTENSION = 'py'

    def emit(self):
        # Access enhancement data
        for func in self.get_functions():
            cfg = self.get_function_cfg(func['name'])
            if cfg:
                # Use CFG for structured code generation
                self._emit_function_with_cfg(func, cfg)
            else:
                # Fallback to linear emission
                self._emit_function_linear(func)

        return self.generate_manifest()

# Create emitter with context
emitter = PythonEmitter(ir_data, '/output', enhancement_context)
manifest = emitter.emit()
```

Configuration

PipelineConfig

```
@dataclass
class PipelineConfig:
    enable_control_flow: bool = True
    enable_type_analysis: bool = True
    enable_semantic_analysis: bool = True
    enable_memory_analysis: bool = True
    enable_optimization: bool = True
    optimization_level: str = "O2"  # O0, O1, O2, O3
    fail_fast: bool = False         # Stop on first failure
    collect_diagnostics: bool = True
    timeout_seconds: float = 0.0    # 0 = no timeout
```

EmitterConfig

```
@dataclass
class EmitterConfig:
    use_enhancements: bool = True
    emit_comments: bool = True
    emit_debug_info: bool = False
    indent_size: int = 4
    line_ending: str = '\n'
    max_line_length: int = 0  # 0 = no limit
```

Command Line Usage

The `emit_code.py` tool now supports enhancement integration:

```
# Basic usage (no enhancements)
python tools/emitters/emit_code.py input.json --target=python

# With enhancement pipeline
python tools/emitters/emit_code.py input.json --target=python --enhance

# With custom optimization level
python tools/emitters/emit_code.py input.json --target=rust --enhance --opt-level=O3

# Verbose output
python tools/emitters/emit_code.py input.json --target=c --enhance -v
```

Enhancement Status

Each enhancement can have the following statuses:

- `NOT_RUN` : Enhancement was not executed
- `SUCCESS` : Enhancement completed successfully
- `PARTIAL` : Enhancement completed with some warnings
- `FAILED` : Enhancement failed (error captured)
- `SKIPPED` : Enhancement was explicitly skipped

Graceful Degradation

The pipeline supports graceful degradation:

1. If an enhancement fails, the pipeline continues with other enhancements
2. Emitters receive partial context and can adapt
3. Original IR is always available as fallback
4. Safe accessors return None/empty when data unavailable

Example:

```python
# Even if type analysis fails, other enhancements work
pipeline = create_pipeline('python')
context = pipeline.run_all_enhancements(ir_data)

# Check what's available
if context.control_flow_data.is_available():
    cfg = context.get_function_cfg('main')
else:
    # Fall back to linear code generation
    pass
```

Testing

Run the integration tests:

```
cd /home/ubuntu/stunir_repo
python -m pytest tests/integration/test_enhancement_integration.py -v
```

Or run directly:

```
python tests/integration/test_enhancement_integration.py
```

Examples

Example 1: Full Pipeline with Rust Target

```python
from tools.integration import create_pipeline

# Load IR
with open('module.json') as f:
    ir_data = json.load(f)

# Run pipeline
pipeline = create_pipeline('rust', 'O2')
context = pipeline.run_all_enhancements(ir_data)

# Check status
print(f"Pipeline complete: {context.get_status_summary()}")

# Use with emitter
from tools.emitters.base_emitter import BaseEmitter

class RustEmitter(BaseEmitter):
    TARGET = 'rust'

    def emit(self):
        for func in self.get_functions():
            # Get memory strategy for Rust (should be 'ownership')
            strategy = self.get_memory_strategy()

            # Get optimized function if available
            opt_func = self.get_optimized_function(func['name'])
            if opt_func:
                func = opt_func

            self._emit_function(func)

        return self.generate_manifest()

emitter = RustEmitter(ir_data, '/output', context)
emitter.emit()
```

Example 2: Disabling Specific Enhancements

```python
from tools.integration import EnhancementPipeline, PipelineConfig

# Only run semantic and control flow analysis
config = PipelineConfig(
    enable_control_flow=True,
    enable_semantic_analysis=True,
    enable_type_analysis=False,
    enable_memory_analysis=False,
    enable_optimization=False
)

pipeline = EnhancementPipeline('python', config)
context = pipeline.run_all_enhancements(ir_data)
```

Example 3: Accessing Enhancement Data in Emitter

```python
class MyEmitter(BaseEmitter):
    def emit(self):
        for func in self.get_functions():
            func_name = func['name']

            # Control flow
            cfg = self.get_function_cfg(func_name)
            loops = self.get_loops(func_name)
            branches = self.get_branches(func_name)

            # Semantic
            func_info = self.lookup_function(func_name)

            # Memory
            strategy = self.get_memory_strategy()

            # Optimization
            hint = self.get_optimization_hint('inline_threshold')

            # Generate code using available data
            if cfg and loops:
                self._emit_structured(func, cfg, loops)
            else:
                self._emit_linear(func)
```

File Structure

```
tools/
├── integration/
│   ├── __init__.py              # Module exports
│   ├── enhancement_context.py   # EnhancementContext class
│   └── enhancement_pipeline.py  # EnhancementPipeline class
├── emitters/
│   ├── base_emitter.py          # BaseEmitter with context support
│   └── emit_code.py             # Updated CLI tool
tests/
└── integration/
    └── test_enhancement_integration.py  # Integration tests
docs/
└── integration/
    └── ENHANCEMENT_INTEGRATION.md  # This documentation
```

Next Steps (Phase 2+)

- Phase 2: Basic Code Generation - Implement body emission using CFGs
- Phase 3: Target Specialization - Language-specific enhancements
- Phase 4: Full Integration - Complete emitter implementations

See Also

- `tools/ir/control_flow.py` - Control flow analysis
- `tools/stunir_types/` - Type system
- `tools/semantic/` - Semantic analysis

- `tools/memory/` - Memory analysis
- `tools/optimize/` - Optimization