02247 Compiler Construction Project Report Group 18

Li, Junrui (s242643)
Li, Leyao (s242598)
Yin, Lingxiao (s242610)

10th May 2025

Contents

Overview

This project significantly extends the Hygge compiler by implementing seven functional enhancements across syntax, semantics, type checking, interpreter, and RISC-V code generation. The main contributions are as follows:

- **Short-circuit logical operators `&&` and `||`** were introduced, enhancing control flow efficiency and semantics.
- **Improved `do...while` loop** support added full operational semantics and code generation for post-condition loops.
- **Support for more than 8 integer arguments** required redesigning stack frame layouts and conforming to the RISC-V calling convention, including precise stack alignment and parameter mapping.
- **Recursive functions with `let rec` and `rec fun`** were implemented, enabling higher-order and self-referential function definitions with full type system integration.
- **Structure copying (shallow and deep)** enhanced the memory model with pointer-aware data duplication, handling nested fields and runtime heap management.
- **Better type inference for `match`** expressions via Least Upper Bound (LUB) calculation allowed more flexible and type-safe pattern matching.
- **Partial evaluation optimization** implemented constant folding, propagation, and dead code elimination, improving execution efficiency.

1 "And" and "Or" with Short-Circuit-Semantics (Medium) — by Yin, Lingxiao

1.1 Formal Specification

1.1.1 Operational Semantics

The following rules define the core reduction semantics for short-circuit logical operators `&&` and `||`.

$$\frac{\langle R \bullet e_1 \rangle \to \langle R' \bullet e_1' \rangle}{\langle R \bullet e_1 \mathbin{\&\&} e_2 \rangle \to \langle R' \bullet e_1' \mathbin{\&\&} e_2 \rangle} \text{(SC-And-Eval1)} \qquad \frac{\langle R \bullet e_1 \rangle \to \langle R' \bullet e_1' \rangle}{\langle R \bullet e_1 \parallel e_2 \rangle \to \langle R' \bullet e_1' \parallel e_2 \rangle} \text{(SC-Or-Eval1)}$$

$$\frac{}{\langle R \bullet \texttt{false} \mathbin{\&\&} e_2 \rangle \to \langle R \bullet \texttt{false} \rangle} \text{(SC-And-False)} \qquad \frac{}{\langle R \bullet \texttt{true} \parallel e_2 \rangle \to \langle R \bullet \texttt{true} \rangle} \text{(SC-Or-True)}$$

$$\frac{}{\langle R \bullet \texttt{true} \mathbin{\&\&} e_2 \rangle \to \langle R \bullet e_2 \rangle} \text{(SC-And-True)} \qquad \frac{}{\langle R \bullet \texttt{false} \parallel e_2 \rangle \to \langle R \bullet e_2 \rangle} \text{(SC-Or-False)}$$

1.2 Implementation

1.2.1 Lexer and Parser

We extend `Lexer.fsl`:

```
| "&&"          { Parser.ANDSC }
| "||"          { Parser.ORSC  }
```

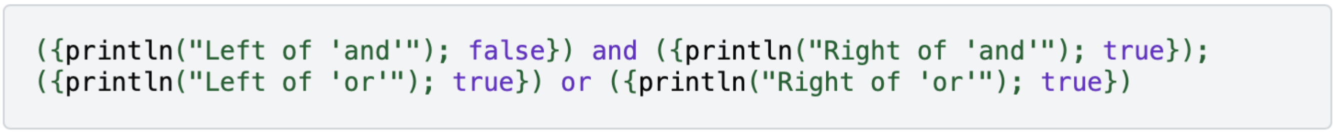
Correspondingly, we declare the new token in `Parser.fsy`, and add a new rule for parsing:

```
%token ANDSC ORSC

let mkBool (ps: IParseState) idx v =
  mkNode(ps, idx, Expr.BoolVal v)

orExpr:
  | orExpr ORSC andExpr  {   // if e1 then true else e2
     mkNode(parseState, 2, Expr.If($1, mkBool parseState 2 true, $3)) }
andExpr:
  | andExpr ANDSC relExpr  {   // if e1 then e2 else false
        mkNode(parseState, 2, Expr.If($1, $3, mkBool parseState 2 false)) }
```

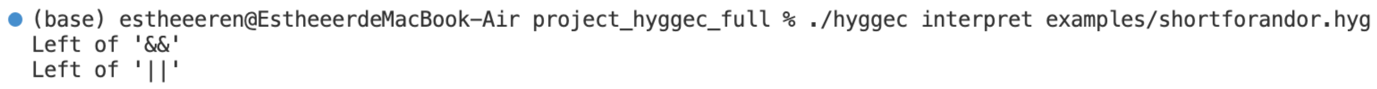
1.2.2 Testing

- tests/lexer/pass/sc.hyg
- tests/parser/pass/sc.hyg
- tests/interpreter/pass/sc.hyg
- tests/typechecker/pass/shortand.hyg
- tests/typechecker/pass/shortor.hyg
- tests/codegen/pass/sc1.hyg

The examples/shortforandor.hyg produced the expected result.



```
({println("Left of 'and'"); false}) and ({println("Right of 'and'"); true});
({println("Left of 'or'"); true}) or ({println("Right of 'or'"); true})
```



```
● (base) estheeeren@EstheeerdeMacBook-Air project_hyggec_full % ./hyggec interpret examples/shortforandor.hyg
  Left of '&&'
  Left of '||'
```

2 a Better "Do. . . While" Loop (Medium) — by Yin, Lingxiao

2.1 Formal Specification

2.1.1 Operational Semantics

We extend the Hygge operational semantics with the following rules:

$$\frac{\langle R \bullet e_1 \rangle \to \langle R' \bullet e_1' \rangle}{\langle R \bullet \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2 \rangle \to \langle R' \bullet \texttt{do}\ \ e_1'\ \ \texttt{while}\ \ e_2 \rangle} \quad \text{(R-DoWhile-Body)}$$

$$\frac{\texttt{isValue}(e_1) \quad \langle R \bullet e_2 \rangle \to \langle R' \bullet e_2' \rangle}{\langle R \bullet \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2 \rangle \to \langle R' \bullet \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2' \rangle} \quad \text{(R-DoWhile-Cond)}$$

$$\frac{\texttt{isValue}(e_1) \quad \texttt{isValue}(e_2) \quad e_2 = \texttt{true}}{\langle R \bullet \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2 \rangle \to \langle R \bullet e_1; \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2 \rangle} \quad \text{(R-DoWhile-True)}$$

$$\frac{\texttt{isValue}(e_1) \quad \texttt{isValue}(e_2) \quad e_2 = \texttt{false}}{\langle R \bullet \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2 \rangle \to \langle R \bullet e_1 \rangle} \quad \text{(R-DoWhile-False)}$$

2.1.2 Typing Rules

We extend the Hygge typing system with the following new rule:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash \texttt{do}\ \ e_1\ \ \texttt{while}\ \ e_2 : \tau} \quad \text{(T-DoWhile)}$$

2.2 Implementation

2.2.1 AST

We extend the hyggec AST with the new named case:

```
    | DoWhile of body: Node<'E,'T>
            * cond: Node<'E,'T>
```

2.2.2 Printer

We extend the `Prettyprinter.fs` with the new named case:

```
    | DoWhile of body: Node<'E,'T>
           * cond: Node<'E,'T>
```

2.2.3 Parser

We add a new rule for parsing:

```
  | DO simpleExpr WHILE simpleExpr  { mkNode(parseState, 1, Expr.DoWhile($2, $4)) }
```

2.2.4 Interpreter

We extend the `ASTUtil.fs` by adding the new case:

```
    | DoWhile(body, cond) ->
        let substBody = subst body var sub
        let substCond = subst cond var sub
        {node with Expr = DoWhile(substBody, substCond)}

    | DoWhile(body, cond) -> Set.union (freeVars body) (freeVars cond)

    | DoWhile(body, cond) -> Set.union (capturedVars body) (capturedVars cond)
```

We extend the `Interpreter.fs` by:

```
match (reduce env body) with
| Some(env1, body') ->
    Some(env1, {node with Expr = DoWhile(body', cond)})
```

Ensures the loop body is reduced first, satisfying the requirement to execute at least once.

```
| None when (isValue cond) ->
    match cond.Expr with
    | BoolVal(true) ->
        Some(env, {node with Expr = DoWhile(origBody, origCond)})
    | BoolVal(false) ->
        Some(env, {node with Expr = body.Expr})
```

When the condition is true, recreates the loop, enabling the loop to continue. When the condition is false, directly returns the value of the loop body, implementing the feature of returning the value from the last execution of the loop body.

2.2.5 Type Checking

We extend the `Typechecker.fs` by:

```
(Ok(tbody), Ok(tcond)) when (isSubtypeOf env tcond.Type TBool) ->
    Ok { Pos = node.Pos; Env = env; Type = tbody.Type; Expr = DoWhile(tbody, tcond) }
```

Ensures the condition expression (tcond) is a boolean type. Sets the entire do-while loop's type to be the same as the body's type (tbody.Type).

2.2.6 Code Generation

We extend the `RISCVCodegen.fs` by:

```
Asm(RV.LABEL(doWhileBodyLabel), "Body␣of␣the␣'do-while'␣loop␣starts␣here")
    ++ (doCodegen env body)
    .AddText(RV.LABEL(doWhileCondLabel), "Condition␣check␣for␣the␣'do-while'␣loop")
    ++ (doCodegen env cond)
    .AddText([
        (RV.BNEZ(Reg.r(env.Target), doWhileBodyLabel),
         "Jump␣back␣to␣loop␣body␣if␣condition␣is␣true")
        (RV.LABEL(doWhileEndLabel), "End␣of␣the␣'do-while'␣loop")
    ])
```

Generate loop body first: Places the body label and generates code for the loop body unconditionally. After the body code, places the condition label and generates code to evaluate the condition. Uses BNEZ instruction to jump back to the body label if the condition is true, creating the loop structure.

2.2.7 Testing

- `tests/parser/pass/dw.hyg`
- `tests/interpreter/pass/dow.hyg`
- `tests/typechecker/pass/dw.hyg`
- `tests/typechecker/fail/dw1.hyg`
- `tests/typechecker/fail/dw2.hyg`
- `tests/codegen/pass/dw.hyg`

Example:

```
//1
do { false} while (false)
//2
let mutable x = 0;
let mutable y = 10;
do {
    x <- x + 1;
    y <- y + 1
}while (x + y < 42);
assert(x + y = 42)
```

3 Pass more than 8 Integer Arguments via The Stack (Hard) — by Li, Junrui

3.1 Stack structure overview

Here is a simple visualization of the revised RISC-V stack frame which could accept more than 8 integer arguments:

```
High addresses
+------------------+
| Caller's frame   |
+------------------+ <-- Previous sp (stored in fp before the call)
| Return address   |
+------------------+
| Saved fp         |
+------------------+ <-- Current fp
| Callee-saved     |
| registers        |
+------------------+
| Local variables  |
+------------------+
| Args to called   |
| functions (>8)   |
+------------------+ <-- Current sp
Low addresses
```

In my implementation, I maintained the memory layout where:

- The frame pointer (fp) serves as the reference point for accessing function parameters and local variables

- Parameters beyond the 8th are stored at positive offsets from fp
- The stack grows downward (toward lower addresses)
- The stack pointer (sp) is adjusted dynamically during function calls

At the function call site, the stack must be prepared to hold extra arguments. The caller is responsible for placing these arguments on the stack before the call and cleaning up afterward.

3.2 Implementation

The work is done within file `RISCVCodegen.fs`.

To represent stack-based variables, I extend the compiler's internal storage model. This requires:

- A new storage type for stack-allocated variables
- Updated code generation for variable access
- Proper stack management in function prologues and epilogues
- Correct parameter passing at call sites

3.2.1 Storage Representation

The first step was extending the `Storage` type in `RISCVCodegen.fs` to represent the variables stored in the stack.

```
/// Storage information for variables.
[<RequireQualifiedAccess; StructuralComparison; StructuralEquality>]
type internal Storage =
    /// The variable is stored in an integerregister.
    | Reg of reg: Reg
    /// The variable is stored in a floating-point register.
    | FPReg of fpreg: FPReg
    /// The variable is stored in memory, in a location marked with a
    /// label in the compiled assembly code.
    | Label of label: string
    /// This variable is stored on the stack, at the given offset (in bytes)
    /// from the memory address contained in the frame pointer (fp) register.
    | Frame of offset: int
```

The `Frame` case stores the byte offset from the frame pointer, allowing variables to be located relative to fp. By using the frame pointer rather than the stack pointer (which changes during function execution), the revised mechanism ensures consistent access to parameters throughout the function's lifetime.

3.2.2 Stack Frame Management and Argument Mapping

The `compileFunction` function was modified to map function arguments to appropriate storage locations while ensuring proper stack frame setup:

```
/// Folder function that assigns storage information to function arguments
let folder (acc: Map<string, Storage>) (i, (var, _tpe)) =
    if i < 8 then
        // First 8 args use registers a0-a7
        acc.Add(var, Storage.Reg(Reg.a((uint)i)))
    else
        // Args beyond 8 use stack locations relative to fp
        // Note: We use (i-8)*4 because each word is 4 bytes
        acc.Add(var, Storage.Frame((i-8)*4))
```

This creates a mapping where:

- First 8 arguments → Registers a0-a7

- 9th argument → Memory at fp+0
- 10th argument → Memory at fp+4
- And so on...

A key challenge was ensuring the frame pointer (fp) was correctly established at function entry. In the function prologue, I save the old frame pointer and set a new one:

- Setting fp at the beginning of each function to create a stable reference point
- Using fp as the base for accessing stack-based variables
- Preserving fp across function calls by saving and restoring it

3.2.3 Variable Access with Stack-Based Parameters

The variable access logic in the `doCodegen` function was updated to handle stack-based variables:

```
// For integer variables stored on stack
| Some(Storage.Frame(offset)) ->
    // Load a variable from the stack at fp+offset
    Asm(RV.LW(Reg.r(env.Target), Imm12(offset), Reg.fp),
        $"Load stack variable '{name}' from fp+{offset}")

// For floating-point variables stored on stack
| Some(Storage.Frame(offset)) ->
    // Load a float variable from the stack at fp+offset
    Asm([ (RV.LW(Reg.r(env.Target), Imm12(offset), Reg.fp),
           $"Load stack variable '{name}' from fp+{offset}")
          (RV.FMV_W_X(FPReg.r(env.FPTarget), Reg.r(env.Target)),
           $"Transfer '{name}' to fp register") ])
```

This generates the appropriate RISC-V instructions to load values from stack locations. The implementation required careful consideration of RISC-V's load word (LW) instruction semantics, ensuring correct offsets from the frame pointer.

3.2.4 Variable Assignment with Stack-Based Storage

Variable assignment also required updates to handle stack-based variables:

```
| Some(Storage.Frame(offset)) ->
    match rhs.Type with
    | t when (isSubtypeOf rhs.Env t TFloat) ->
        rhsCode.AddText(RV.FSW_S(FPReg.r(env.FPTarget), Imm12(offset), Reg.fp),
                       $"Assignment to stack variable {name} at fp+{offset}")
    | _ ->
        rhsCode.AddText(RV.SW(Reg.r(env.Target), Imm12(offset), Reg.fp),
                       $"Assignment to stack variable {name} at fp+{offset}")
```

This generates RISC-V store instructions to update variables on the stack. The implementation handles different variable types appropriately, using SW for integer values.

3.2.5 Function Call Implementation and Stack Alignment

The workflow for this part goes like this: adjusts the stack pointer to make room for extra arguments, then stores each argument at the appropriate stack offset. After the function call returns, we clean up the stack by restoring the stack pointer.

```
// Function that stores extra arguments (beyond the 8th) on the stack
let storeExtraArg (acc: Asm) (i: int) =
    acc.AddText(RV.SW(Reg.r(env.Target + (uint i) + 1u), Imm12((i - 8) * 4), Reg.sp),
               $"Store extra function call argument {i+1} at sp+{(i - 8) * 4}")

// Determine how many arguments go in registers and how many on stack
let regArgsCount = min 8 args.Length
let extraArgsCount = max 0 (args.Length - 8)

// Calculate padding needed for 16-byte alignment
let alignmentPadding = (extraArgsCount % 4) * 4

// Code to store extra arguments on the stack
let stackArgsStoreCode =
    if extraArgsCount > 0 then
        // Adjust stack pointer to make room for extra arguments + alignment
        Asm(RV.ADDI(Reg.sp, Reg.sp, Imm12(-4 * extraArgsCount - alignmentPadding)),
            $"Adjust␣stack␣pointer␣for␣{extraArgsCount}␣extra␣arguments␣with␣alignment")
        ++ (List.fold storeExtraArg (Asm()) [8..(args.Length-1)])
    else
        Asm() // No extra arguments to store
```

This implementation addresses several challenges:

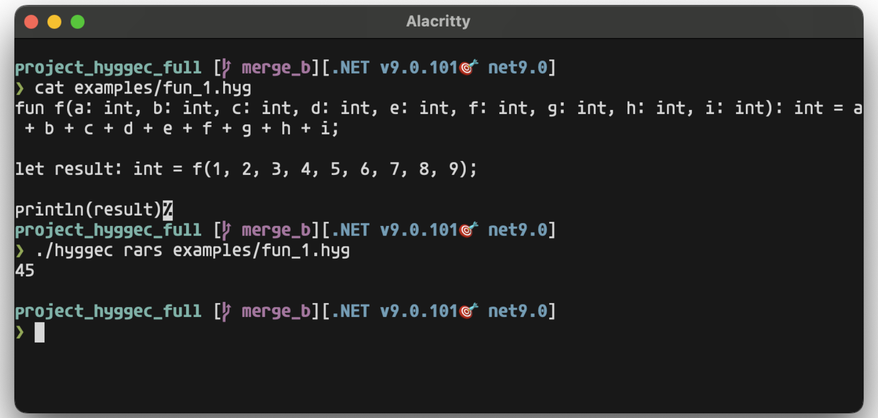
1. **Stack Alignment**: RISC-V requires the stack to be 16-byte aligned at function calls. I calculate the the appropriate padding to maintain this alignment.

2. **Parameter Ordering**: Arguments are stored in reverse order to match the expectations of the RISC-V calling convention.

3. **Register Spilling**: With more complex functions, the register pressure increases. My implementation handles this by carefully managing register allocation and using the stack for variables that cannot be kept in registers.

3.3 Evaluation

The implementation successfully extends the Hygge compiler to support functions with more than 8 integer parameters. But just a notion here, the current implementation doesn't not support code generation path with ANF(the -a parameter). Two example test cases `fun_1.hyg`(9 integer arguments), `fun_2.hyg`(10 integer ooooarguments) are given at `/project_dir/examples/`.



```
project_hyggec_full [ merge_b][.NET v9.0.101🎯 net9.0]
❯ cat examples/fun_1.hyg
fun f(a: int, b: int, c: int, d: int, e: int, f: int, g: int, h: int, i: int): int = a
 + b + c + d + e + f + g + h + i;

let result: int = f(1, 2, 3, 4, 5, 6, 7, 8, 9);

println(result)%
project_hyggec_full [ merge_b][.NET v9.0.101🎯 net9.0]
❯ ./hyggec rars examples/fun_1.hyg
45

project_hyggec_full [ merge_b][.NET v9.0.101🎯 net9.0]
❯
```

Figure 1: Test the revised RISCVCodegen.fs with 9 integer arguments

4 Recursive Functions (Hard) — by Li, Leyao

4.1 Formal Specification

4.1.1 Syntax

We extend the Hygge grammar with the following new rule:

$$\begin{array}{lrcl} \text{Expression} & e & ::= & \ldots \\ & & \mid & \texttt{let rec } x : t = e; e' \end{array}$$

We also introduce syntactic sugar to simplify the definition of recursive functions. The following syntactic form:

$$\texttt{rec fun } \mathit{name}(x_1 : t_1, \ldots, x_n : t_n) : t \quad = \quad \mathit{body};\ e'$$

is desugared into:

$$\texttt{let rec } \mathit{name} : (t_1 \times \ldots \times t_n \to t) \quad = \quad (\lambda(x_1 : t_1, \ldots, x_n : t_n) \to \mathit{body});\ e'$$

We additionally support a variant form where the function body itself is a lambda expression:

$$\texttt{rec fun } \mathit{name}(x_1 : t_1, \ldots) : t \quad = \quad \texttt{fun } (y_1 : t'_1, \ldots) \to \mathit{expr};\ e'$$

which is desugared into:

$$\texttt{let rec } \mathit{name} : (t_1 \times \ldots \to t) \quad = \quad (\lambda(y_1 : t'_1, \ldots) \to \mathit{expr});\ e'$$

This extended sugar makes it convenient to write recursive and higher-order functions in a concise and expressive way.

4.1.2 Operational Semantics

We extend the definition of substitution $e[x \mapsto e']$ with the following new cases:

$$\begin{array}{rcl} (\texttt{let rec } x : t = e_1; e_2)[x \mapsto e'] & = & \texttt{let rec } x : t = e_1; e_2 \\ (\texttt{let rec } y : t = e_1; e_2)[x \mapsto e'] & = & \texttt{let rec } y : t = e_1[x \mapsto e'];\ e_2[x \mapsto e'] \quad (y \neq x) \end{array}$$

We extend the Hygge operational semantics with the following rules:

$$\frac{\langle R \bullet e \rangle \to \langle R' \bullet e' \rangle}{\langle R \bullet \texttt{let rec } x : t = e; e' \rangle \to \langle R' \bullet \texttt{let rec } x : t = e'; e' \rangle} \quad \text{(R-LetRec-Eval-Init)}$$

$$\frac{v' = v[x \mapsto (\texttt{let rec } x : t = v; x)]}{\langle R \bullet \texttt{let rec } x : t = v; e \rangle \to \langle R \bullet e[x \mapsto v'] \rangle} \quad \text{(R-LetRec-Subst)}$$

The first rule (R-LetRec-Eval-Init) reduces the initialization expression e inside a recursive let-binding. The second rule (R-LetRec-Subst) applies once the initialization expression has reduced to a value v, substituting x in e' with v', where v' is a copy of v in which any reference to x is replaced by a self-reference to the recursive binding. This allows recursive functions to refer to themselves during evaluation.

4.1.3 Typing Rules

We extend the Hygge typing system with the following new rule:

$$\frac{\Gamma \vdash t \triangleright T \qquad \Gamma' = \left(\begin{array}{l}\Gamma \text{ with } \mathit{Vars} + (x \mapsto T) \\ \text{and } \mathit{Mutables} \setminus \{x\}\end{array}\right) \qquad \Gamma' \vdash \lambda(x_1 : t_1, \ldots) \to e_1 : T \quad \Gamma' \vdash e_2 : T'}{\Gamma \vdash \texttt{let rec } x : t = (\lambda(x_1 : t_1, \ldots) \to e_1); e_2 : T'} \quad \text{(T-LetRec)}$$

The new typing rule [T-LetRec] differs from [T-Let-T2] in that the initialization expression for the variable x is typed under an extended environment Γ', where x is already bound with type T. This allows the expression to refer to x recursively. Additionally, the initialization expression must be a lambda abstraction, ensuring that x has a function type and preventing invalid definitions such as `let rec x : t = x`.

4.2 Implementation

Outline how you implemented this improvement/extension to Hygge and `hyggec`. Avoid long code snippets: you can put a brief outline here, and refer to the relevant file in the submitted source code.

4.2.1 AST

We extend the `hyggec` AST with the new named case:

```
and Expr<'E,'T> = // ...
    /// Let-rec binding to define a recursive function with a given name and type.
    /// The function can refer to itself recursively in the init expression.
    | LetRec of name: string
              * tpe: PretypeNode
              * init: Node<'E,'T>
              * scope: Node<'E,'T>
```

4.2.2 Lexer and Parser

We extend `Lexer.fsl` with a new rule for producing the `REC` token:

```
| "rec"         { Parser.REC }
```

Correspondingly, we declare the new `REC` token in `Parser.fsy`, along with other keywords:

```
%token LET REC TYPE MUTABLE
```

We then add a new parsing rule to handle recursive function definitions:

```
| LET REC variable COLON pretype EQ expr SEMI expr {
    mkNode(parseState, 1, Expr.LetRec($3, $5, $7, $9))
}
```

To make the syntax more concise for users, we also implement syntactic sugar for recursive function declarations of the form `rec fun name(...):...  = ...`, which desugars into a `let rec` binding:

```
| REC FUN variable parenArgTypesSeq COLON pretype EQ expr SEMI expr {
    let (_, argTypes) = List.unzip $4
    mkNode(parseState, 1,
        Expr.LetRec(
          $3,
          mkPretypeNode(parseState, 6, Pretype.TFun(argTypes, $6)),
          mkNode(parseState, 8, Lambda($4, $8)),
          $10
        )
    )
}
```

This syntactic sugar allows programmers to write recursive functions in a natural and concise style, and internally expands to a `LetRec` AST node with a lambda term as the initializer.

4.2.3 Interpreter

We extend the function `subst` in `ASTUtil.fs` by adding the new case:

```
  | LetRec(vname, tpe, init, scope) when vname = var ->
      // Do not substitute 'var' in init or scope
      {node with Expr = LetRec(vname, tpe, init, scope)}
```

This case ensures that if the variable being substituted is the one bound by the `let rec`, then substitution is blocked both in the scope and in the initializer. This follows the substitution rules in Definition 2, preventing accidental capture in recursive bindings.

We extend the function `reduce` in `Interpreter.fs` by adding the new case:

```
  | LetRec(name, tpe, init, scope) ->
      match reduce env init with
      | Some(env', init') ->
          Some(env', {node with Expr = LetRec(name, tpe, init', scope)})

      | None when isValue init ->
          match init.Expr with
          | Lambda(args, body) ->
              // [R-LetRec-Subst]: substitute self-reference
              let recTerm =
                  {node with Expr = LetRec(name, tpe, init, {node with Expr = Var(name)})}
              let newBody = ASTUtil.subst body name recTerm
              let newInit = {init with Expr = Lambda(args, newBody)}
              Some(env, {node with Expr = ASTUtil.subst scope name newInit.Expr})
          | _ -> None // Only lambda terms are allowed
      | None -> None
```

This implementation covers both evaluation rules:

- **[R-LetRec-Eval-Init]**: evaluates the initializer expression first, reducing it if not yet a value.
- **[R-LetRec-Subst]**: if the initializer is a lambda, substitutes recursive references within it to construct v', and then replaces the bound variable in the scope.

This also enforces that only lambda terms are valid initializers for recursive definitions.

4.2.4 Type Checking

We extend the function `typer` in `Typechecker.fs` by adding the new case:

```
| LetRec(name, tpe, init, scope) ->
    letRecTyper node.Pos env name tpe init scope
```

We implement the `[T-LetRec]` typing rule in a new helper function `letRecTyper`:

```
and internal letRecTyper pos (env: TypingEnv) (name: string)
    (tpe: PretypeNode) (init: UntypedAST) (scope: UntypedAST): TypingResult =

    match (resolvePretype env tpe) with
    | Ok(funType) ->
        // Extend environment with the recursive function being defined
        let envVars2 = env.Vars.Add(name, funType)
        let envMutVars2 = env.Mutables.Remove(name)
        let env2 = {env with Vars = envVars2; Mutables = envMutVars2}

        // Typecheck the initializer under the extended environment
        match (typer env2 init) with
        | Ok(tinit) ->
            match (tinit.Type, funType) with
            | (TFun(_, _), TFun(_, _)) when isSubtypeOf env2 tinit.Type funType ->
```

```
                // Typecheck the scope using the recursive binding
                match (typer env2 scope) with
                | Ok(tscope) ->
                    let expr = LetRec(name, tpe, tinit, tscope)
                    Ok { Pos = pos; Env = env; Type = tscope.Type; Expr = expr }
                | Error(es) -> Error(es)
            | (TFun(_, _), TFun(_, _)) ->
                Error([(pos, $"Function␣'%s{name}'␣of␣type␣%O{funType}␣"
                        + $"initialized␣with␣function␣of␣incompatible␣type␣%O{tinit.Type}")])
            | _ ->
                Error([(init.Pos, $"let␣rec␣initialization␣must␣be␣a␣function,␣"
                        + $"found␣expression␣of␣type␣%O{tinit.Type}")])
        | Error(es) -> Error(es)
    | Error(es) -> Error(es)
```

This implementation follows the [T-LetRec] typing rule, ensuring:

- An extended environment Γ' is created, binding the recursive variable x to its declared function type.
- The initializer is typechecked under Γ' and must be a lambda (function).
- The scope is typechecked under the same extended environment.
- The function type of the initializer must be a subtype of the declared type.
- Only functions are allowed as recursive initializers — all other expressions are rejected.

4.2.5 Code Generation

We extend the function `doCodegen` in `RISCVCodegen.fs` by adding the new case:

```
| LetRec(name, _, {Node.Expr = Lambda(args, body); Node.Type = TFun(targs, _)}, scope) ->
    // Generate an assembly label for the function
    let funLabel = Util.genSymbol $"fun_%s{name}"

    // Extract argument names and zip with types
    let (argNames, _) = List.unzip args
    let argNamesTypes = List.zip argNames targs

    // Extend the environment so the function name maps to its label
    let varStorage2 = env.VarStorage.Add(name, Storage.Label(funLabel))

    // Compile the body of the function with the updated environment
    let bodyCode = compileFunction argNamesTypes body {env with VarStorage = varStorage2}

    // Wrap the function body in a labeled code block
    let funCode =
        (Asm(RV.LABEL(funLabel), $"Code␣for␣recursive␣function␣'%s{name}'")
         ++ bodyCode).TextToPostText

    // Compile the scope of the let-rec with access to the new label
    (doCodegen {env with VarStorage = varStorage2} scope)
        ++ funCode
```

This implementation handles the code generation for recursive functions defined using `let rec`. A unique assembly label is created for the function body and stored in the variable environment, allowing recursive calls within the function. The function body is compiled with this extended environment, and the label/code block is appended to the final text segment using `TextToPostText`. The scope of the `let rec` is then compiled with the same environment, ensuring the recursive function is correctly accessible during execution.

4.2.6 Testing

We added the following test files under the `tests/` directory:

- `tests/lexer/pass/008-rec.hyg`
- `tests/parser/pass/021-rec.hyg`
- `tests/interpreter/pass/027-rec.hyg`
- `tests/typechecker/pass/027-rec.hyg`

- `tests/typechecker/fail/022-rec.hyg`
- `tests/codegen/pass/026-rec.hyg`

We also provide a representative test case `tests/codegen/pass/026-rec.hyg`, which test:

- **Standard recursion**: The `fibonacci` function tests classic recursive behavior.
- **Tail recursion**: The `tailFactorial` function tests optimized recursion using an accumulator.
- **Recursive function composition**: The `factorial` function tests invoking one recursive function (`tailFactorial`) from another.
- **Multiple recursive calls**: `fibonacci` contains two recursive calls in one expression (`fibonacci(n + -1) + fibonacci(n + -2)`).
- **Call chains**: `factorial` calls `tailFactorial`, verifying recursive delegation and argument evaluation.
- **Evaluation rule** [**R-LetRec-Eval-Init**]: Each recursive definition triggers this rule to initialize the function binding.
- **Substitution rule** [**R-LetRec-Subst**]: The rule is validated when recursive functions refer to themselves inside lambda bodies.
- **Typing rule** [**T-LetRec**]: The functions are correctly typed with declared functional types, including multi-parameter lambdas.
- **Syntactic sugar `rec fun`**: The test validates support for both single- and multi-parameter recursive function declarations using syntactic sugar.

Here is a sample test from `026-rec.hyg`:

```
rec fun fibonacci(n: int): int =
    if (n <= 1) then n
    else fibonacci(n + -1) + fibonacci(n + -2);

rec fun tailFactorial(n: int, acc: int): int =
    if (n == 0) then acc
    else tailFactorial(n + -1, acc * n);

rec fun factorial(n: int): int =
    tailFactorial(n, 1);

assert(fibonacci(5) == 5);
assert(tailFactorial(5, 1) == 120);
assert(factorial(6) == 720);
```

5 Extend Hygge with Copying of Structures (Hard) — by Yin, Lingxiao

Since the implementations of `deep copy` and `shallow copy` only differ in certain parts of the code, the term `copy` will be used in the report to refer to the common parts, while the differing parts will be described separately as `deepcopy` and `shallowcopy`.

5.1 Formal Specification

5.1.1 Operational Semantics

We extend the Hygge operational semantics with the following rules:

$$\texttt{copy}(e)[x \mapsto e'] \quad = \quad \texttt{copy}(e[x \mapsto e'])$$

$$\frac{\langle R \bullet e\rangle \to \langle R' \bullet e'\rangle}{\langle R \bullet \texttt{copy}(e)\rangle \;\to\; \langle R' \bullet \texttt{copy}(e')\rangle}\ \text{[R-Copy-Arg]}$$

$$\frac{\begin{array}{c}\texttt{isValue}(e) \qquad e = \texttt{Pointer}(a) \\ R = \langle H, P, M\rangle \qquad P(a) = [f_1, \ldots, f_n] \qquad v_i = H(a+i-1) \quad \forall i \in \{1,\ldots,n\} \\ (H', b) = \texttt{allocHeap}(H, \texttt{deepCopyValues}(R, [v_1,\ldots,v_n]))\end{array}}{\langle R \bullet \texttt{deepcopy}(e)\rangle \;\to\; \big\langle \langle H', P \cup \{b \mapsto [f_1,\ldots,f_n]\}, M\rangle \bullet \texttt{Pointer}(b)\big\rangle}\ \text{[R-DeepCopy-Struct]}$$

$$\frac{\begin{array}{c}\texttt{isValue}(e) \qquad e = \texttt{Pointer}(a) \qquad R = \langle H, P, M\rangle \qquad P(a) = [f_1, \ldots, f_n] \\ v_i = H(a+i-1) \quad \forall i \in \{1,\ldots,n\} \qquad (H', b) = \texttt{allocHeap}(H, [v_1,\ldots,v_n])\end{array}}{\langle R \bullet \texttt{shallowcopy}(e)\rangle \;\to\; \big\langle \langle H', P \cup \{b \mapsto [f_1,\ldots,f_n]\}, M\rangle \bullet \texttt{Pointer}(b)\big\rangle}\ \text{[R-ShallowCopy-Struct]}$$

5.1.2 Typing Rules

We extend the Hygge typing system with the following new rule:

$$\frac{\Gamma \vdash e : T \qquad \text{isStructType}(T)}{\Gamma \vdash \texttt{copy}(e) : T}\ \text{[T-Copy]}$$

5.2 Implementation

5.2.1 AST

We extend the `hyggec` AST with the new named case:

```
    | Copy of arg: Node<'E,'T>
```

5.2.2 Printer

We extend the `Prettyprinter.fs` with the new named case:

```
    | Copy(arg) ->
        mkTree "Copy" node [("arg", formatASTRec arg)]
```

5.2.3 Lexer and Parser

We extend `Lexer.fsl` with:

```
| "deepcopy"       { Parser.DEEPCOPY}
| "shallowcopy"       { Parser.SHALLOWCOPY}
```

We declare in `Parser.fsy`, and add a new rule for parsing subtractions:

```
%token DEEPCOPY  SHALLOWCOPY

| DEEPCOPY LPAR simpleExpr RPAR      { mkNode(parseState, 1, Expr.DeepCopy($3)) }
| SHALLOWCOPY LPAR simpleExpr RPAR      { mkNode(parseState, 1, Expr.ShallowCopy($3)) }
```

5.2.4 Interpreter

We extend `ASTUtil.fs` by adding the new case

```
    | Copy(arg) ->
        {node with Expr = Copy(subst arg var sub)}

    | Copy(arg) -> freeVars arg

    | Copy(arg) -> capturedVars arg
```

We extend `Interpreter.fs` by adding the new case:

```
    | ShallowCopy(arg) ->
        match (reduce env arg) with
        | Some(env', arg') ->
            Some(env', {node with Expr = ShallowCopy(arg')})
        | None when isValue arg ->
            match arg.Expr with
            | Pointer(addr) ->
                match (env.PtrInfo.TryFind addr) with
                | Some(fields) ->
                // Get field values from source structure
                    let fieldValues =
                        List.mapi (fun i _ -> env.Heap[addr + (uint i)]) fields
                // Allocate memory for new structure
                    let (heap', baseAddr) = heapAlloc env.Heap fieldValues
                // Update pointer information for new structure
                    let ptrInfo' = env.PtrInfo.Add(baseAddr, fields)
                    Some({env with Heap = heap'; PtrInfo = ptrInfo'},
                         {node with Expr = Pointer(baseAddr)})
                | None -> None
            | _ -> None
        | None -> None
```

For `shallowcopy`, first attempt to evaluate the argument expression. If the argument can be reduced further, continue evaluation recursively. Once the argument is fully evaluated to a value:

- Verify it's a pointer to a structure
- Look up the structure's field information in the runtime environment
- Extract all field values directly from the heap, without modifying them
- Allocate new memory space for a duplicate structure
- Copy all original field values into the new structure
- Update the runtime environment to record the new structure's metadata
- Return a pointer to the newly created structure

```
// Process each field recursively to handle nested structs
for i = 0 to (fieldValues.Length - 1) do
    let fieldVal = fieldValues.[i]
    match fieldVal.Expr with
    | Pointer(fieldAddr) when finalEnv.PtrInfo.ContainsKey(fieldAddr) ->
        // Recursively copy nested struct
        let (envAfterCopy, newAddr) = deepCopyStruct finalEnv fieldAddr
        // Update the copied struct field to point to the new nested struct
        let updatedHeap = envAfterCopy.Heap.Add(baseAddr + (uint i),
                                    {fieldVal with Expr = Pointer(newAddr)})
        finalEnv <- {envAfterCopy with Heap = updatedHeap}
```

For `deepcopy`, previous process is the same as `shallowcopy`. When the main recursive copying process begins:

- Create a helper function deepCopyStruct that takes a source address and returns a new address
- Start by copying all field values from the source structure
- Allocate new memory for the duplicate structure
- For each field in the structure: If the field is a pointer to another structure, recursively copy that structure. Update the field in the new structure to point to the recursively copied structure. Otherwise, keep the field value as is

- Return the updated environment and address of the completely copied structure

5.2.5 Type Checking

We extend `Typechecker.fs` by adding the new case:

```
    | Copy(arg) ->
        match (typer env arg) with
        | Ok(targ) ->
            match (expandType env targ.Type) with
            | TStruct(_) ->
                Ok { Pos = node.Pos; Env = env; Type = targ.Type; Expr = Copy(targ) }
            | _ ->
                Error([(node.Pos, $"copy operation: expected argument of struct type, found 
    %O{targ.Type}")])
        | Error(es) -> Error(es)
```

5.2.6 Code Generation

For `shallowcopy`, We extend `RISCVCodegen.fs` by adding the new case:

```
let structAllocCode =
                (beforeSysCall [Reg.a0] [])
                    .AddText([
                        (RV.LI(Reg.a0, fieldCount * 4),
                        "Allocate memory space for the struct copy (in bytes)")
                        (RV.LI(Reg.a7, 9), "RARS syscall: Sbrk")
                        (RV.ECALL, "")
                        (RV.MV(Reg.r(env.Target + 1u), Reg.a0),
                        "Save system call result (struct memory address) to temporary register")
                    ])
```

Uses a system call to allocate memory for the new structure.

```
for i in 0 .. (fieldCount - 1) do
                    let fieldType = snd fields.[i]
                    if isSubtypeOf arg.Env fieldType TFloat then
                    // Floating point fields
                        copyCode <- copyCode.AddText([
                            (RV.FLW_S(FPReg.r(env.FPTarget), Imm12(i * 4), Reg.r(env.Target)),
                            $"Load field %d{i} from source struct (floating point)")
                            (RV.FSW_S(FPReg.r(env.FPTarget), Imm12(i * 4), Reg.r(env.Target + 1u)),
                            $"Copy field %d{i} to target struct (floating point)")
                    ])
                    else if not (isSubtypeOf arg.Env fieldType TUnit) then
                    // Integer and other type fields
                        copyCode <- copyCode.AddText([
                            (RV.LW(Reg.r(env.Target + 2u), Imm12(i * 4), Reg.r(env.Target)),
                            $"Load field %d{i} from source struct")
                            (RV.SW(Reg.r(env.Target + 2u), Imm12(i * 4), Reg.r(env.Target + 1u)),
                            $"Copy field %d{i} to target struct")
                        ])
```

Directly copies each field value, including pointer values.

For `deepcopy`, We extend `RISCVCodegen.fs` by adding the new case:

```
// The recursive function that implements deep copy
let rec generateDeepCopyCode (sourceReg: uint) (targetReg: uint) (structType: Type) : Asm =
```

```
| TStruct(_) ->
        // Handle nested structs
        let nestedSourceReg = sourceReg + 2u
        let nestedTargetReg = targetReg + 3u
        // Load pointer to nested struct
        code <- code.AddText([
            (RV.LW(Reg.r(nestedSourceReg), Imm12(i * 4), Reg.r(sourceReg)),
            $"Load nested struct pointer from field {fieldName}")
        ])
        // Save current register state
        code <- code.AddText([
            (RV.SW(Reg.r(sourceReg), Imm12(-4), Reg.sp), "Save source struct address")
            (RV.SW(Reg.r(targetReg), Imm12(-8), Reg.sp), "Save target struct address")
            (RV.ADDI(Reg.sp, Reg.sp, Imm12(-8)), "Adjust stack pointer")
        ])
        // Recursively copy nested struct
        code <- code ++ generateDeepCopyCode nestedSourceReg nestedTargetReg fieldType
        // Restore register state
        code <- code.AddText([
            (RV.ADDI(Reg.sp, Reg.sp, Imm12(8)), "Restore stack pointer")
            (RV.LW(Reg.r(sourceReg), Imm12(-4), Reg.sp), "Restore source struct address")
            (RV.LW(Reg.r(targetReg), Imm12(-8), Reg.sp), "Restore target struct address")
        ])
        // Store new nested struct address in current struct
        code <- code.AddText([
            (RV.SW(Reg.r(nestedTargetReg), Imm12(i * 4), Reg.r(targetReg)),
            $"Store new nested struct address to field {fieldName}")
        ])
```

First, takes a source structure address and generates code to allocate memory for a new structure copy and process each field according to its type. For Basic type (integer, float): generates direct value copy instructions. For Nested structure: recursively generates code to:

- Load the nested structure pointer
- Save current context on stack
- Perform complete deep copy of the nested structure
- Restore context from stack
- Update field in parent structure to point to the new copy

The recursive function design ensures every structure in the object graph gets independently copied, with all internal pointers appropriately reconnected to new copies.

5.2.7 Testing

- `tests/lexer/pass/copy.hyg`
- `tests/parser/pass/deepcopy.hyg`
- `tests/parser/pass/shallowcopy.hyg`
- `tests/interpreter/pass/deepcopy.hyg`
- `tests/interpreter/pass/shallowcopy.hyg`
- `tests/typechecker/pass/deepcopy.hyg`
- `tests/typechecker/pass/shallowcopy.hyg`
- `tests/typechecker/fail/deepcopy.hyg`
- `tests/typechecker/fail/shallowcopy.hyg`
- `tests/codegen/pass/deepcopy.hyg`
- `tests/codegen/pass/shallowcopy.hyg`

`Deepcopy` example:

```
let s1 = struct {f = struct {g = 0.0f}};
let s2 = deepcopy(s1);
s1.f.g <- 1.0f;
assert(s1.f.g = 1.0f);
assert(s2.f.g = 0.0f)
```

For `Shallowcopy` example:

```
let s1 = struct {f = struct {g = 0.0f}};
let s2 = shallowcopy(s1);
s1.f.g <- 1.0f;
assert(s1.f.g = 1.0f);
assert(s2.f.g = 1.0f)
```

6 Better Inference of the Pattern Matching Result Type (Hard) — by Li, Leyao

6.1 Formal Specification

6.1.1 Type System Extensions

In this project, we extend the Hygge type system by introducing the concept of the Least Upper Bound (LUB), which is essential for improving type inference in pattern matching expressions.

For two types T_1 and T_2, their least upper bound $LUB(T_1, T_2) = T$ represents a type that satisfies the following conditions: T_1 is a subtype of T ($T_1 <: T$), T_2 is a subtype of T ($T_2 <: T$), and there exists no type S such that $T_1 <: S <: T$ and $T_2 <: S <: T$. Formally, we define:

$$LUB(T_1, T_2) = T \Leftrightarrow (T_1 <: T) \wedge (T_2 <: T) \wedge \forall S.((T_1 <: S \wedge T_2 <: S) \Rightarrow (T <: S))$$

For basic types in Hygge (bool, int, float, string, unit), the LUB of identical types is the type itself, while the LUB of distinct basic types is undefined:

$$\begin{array}{l} LUB(T, T) = T \quad \text{(for all basic types } T\text{)} \\ LUB(T_1, T_2) = \text{undefined} \quad \text{(when } T_1 \neq T_2 \text{ and both are basic types)} \end{array}$$

For types in a subtype relationship, the LUB is the supertype:

$$\begin{array}{l} \text{If } T_1 <: T_2, \text{ then } LUB(T_1, T_2) = T_2 \\ \text{If } T_2 <: T_1, \text{ then } LUB(T_1, T_2) = T_1 \end{array}$$

For union types, given:

$$\begin{array}{rcl} T_1 & ::= & \text{union } \{l_1 : S_1; l_2 : S_2; \ldots\} \\ T_2 & ::= & \text{union } \{l_2 : R_2; l_3 : R_3; \ldots\} \end{array}$$

Their LUB is a union type containing all labels, where types for shared labels are computed as the LUB of the corresponding types:

$$LUB(T_1, T_2) = \text{union } \{l_1 : S_1; l_2 : LUB(S_2, R_2); l_3 : R_3; \ldots\}$$

For structure types, given:

$$\begin{array}{rcl} T_1 & ::= & \text{struct } \{f_1 : S_1; f_2 : S_2; \ldots\} \\ T_2 & ::= & \text{struct } \{f_2 : R_2; f_3 : R_3; \ldots\} \end{array}$$

Their LUB contains only common fields, with field types computed as the LUB of the corresponding types:

$$LUB(T_1, T_2) = \text{struct } \{f_2 : LUB(S_2, R_2); \ldots\}$$

If two structure types share no common fields, their LUB does not exist.

For function types with the same number of parameters:

$$\begin{array}{rcl} T_1 & ::= & (A_1,\ A_2,\ \ldots) \to R_1 \\ T_2 & ::= & (B_1,\ B_2,\ \ldots) \to R_2 \end{array}$$

Since function parameters are contravariant and return types are covariant, the LUB is computed as:

$$LUB(T_1, T_2) = (GLB(A_1, B_1), GLB(A_2, B_2), \ldots) \to LUB(R_1, R_2)$$

where GLB denotes the Greatest Lower Bound. If the GLB of any parameter types does not exist, the LUB of the function types does not exist either.

For multiple types, the LUB calculation exhibits commutativity and associativity properties:

$$\begin{array}{l} LUB(T_1, T_2) = LUB(T_2, T_1) \quad \text{(commutativity)} \\ LUB(T_1, LUB(T_2, T_3)) = LUB(LUB(T_1, T_2), T_3) \quad \text{(associativity)} \end{array}$$

This allows computing the LUB of multiple types in any order:

$$LUB(T_1, T_2, \ldots, T_n) = LUB(T_1, LUB(T_2, LUB(\ldots, T_n)))$$

6.1.2 Typing Rules

In this section, we describe how to modify the type inference rules for pattern matching expressions. The original type checking system for pattern matching in the Hygge compiler had a limitation where it used only the type of the first pattern matching case as the result type of the entire expression. This approach is restrictive and can lead to type errors when subsequent cases have compatible but not identical types.

Pattern matching expressions in Hygge have the following syntax:

$$\begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \quad \texttt{label}_1\{x_1\} \to e_1 \\ \quad \texttt{label}_2\{x_2\} \to e_2 \\ \quad \ldots \\ \quad \texttt{label}_n\{x_n\} \to e_n \end{array}$$

Our improved type inference rule utilizes the Least Upper Bound (LUB) to determine the result type of the entire pattern matching expression. The new rule is formalized as:

$$\frac{\Gamma \vdash e_1 : T_1, \Gamma \vdash e_2 : T_2, \ldots, \Gamma \vdash e_n : T_n}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \ldots : LUB(T_1, T_2, \ldots, T_n)}$$

This rule states that if each continuation expression e_i has type T_i in context Γ, then the type of the entire pattern matching expression is the least upper bound of all these continuation expression types.

When the pattern matching expression is type-checked, we collect the types of all continuation expressions and apply the 'computeLUBMany' function to determine the result type:

$$\text{typeCheckPatternMatch}(T_1, T_2, \ldots, T_n) = \text{computeLUBMany}([T_1, T_2, \ldots, T_n])$$

6.2 Implementation

The implementation of this extension required modifications to two key files in the Hygge compiler. In TypeInference.fs, I implemented the core LUB calculation functions: for two types, for multiple types, and specifically for pattern matching expressions. In Typechecker.fs, I modified the pattern matching type checking logic to use these new functions, replacing the original approach that only used the first case's type. These changes allowed for more flexible and expressive pattern matching while maintaining type safety.

6.2.1 Type Inference

We implement the LUB calculation in the TypeInference module to support the improved pattern matching result type inference:

```
/// Computes the least upper bound (LUB) of two types
let rec computeLUB (t1: Type) (t2: Type): Type =
    if isSubtypeOf t1 t2 then t2
    elif isSubtypeOf t2 t1 then t1
    else
        match t1, t2 with
        | TUnion(cases1), TUnion(cases2) ->
            let allLabels =
                List.map fst cases1 @ List.map fst cases2
                |> List.distinct
            let mergedCases =
                allLabels
                |> List.map (fun label ->
                    let type1Opt = List.tryFind (fun (l, _) -> l = label) cases1 |> Option.map snd
                    let type2Opt = List.tryFind (fun (l, _) -> l = label) cases2 |> Option.map snd
                    match type1Opt, type2Opt with
                    | Some t1, Some t2 -> (label, computeLUB t1 t2)
                    | Some t1, None -> (label, t1)
                    | None, Some t2 -> (label, t2)
                    | None, None -> failwith "Impossible: label not found in either union"
                )
            TUnion(mergedCases)
```

The core of our implementation is the recursive `computeLUB` function that handles different type combinations. For union types, we merge all labels from both types, recursively computing the LUB for matching labels.

For structure types, we find common fields and compute their LUB:

```
| TStruct(fields1), TStruct(fields2) ->
    // Get field names from both structs
    let fieldNames1 = List.map fst fields1 |> Set.ofList
    let fieldNames2 = List.map fst fields2 |> Set.ofList
    // Find common field names
    let commonFieldNames = Set.intersect fieldNames1 fieldNames2

    // Condition 1: LUB exists only if there's at least one common field
    if Set.isEmpty commonFieldNames then
        failwith $"LUB does not exist: struct types {t1} and {t2} have no common fields"
    else
        // Compute LUB for each common field
        let fieldLUBResults =
            commonFieldNames
            |> Set.toList
            |> List.map (fun fieldName ->
                let type1 = List.find (fun (name, _) -> name = fieldName) fields1 |> snd
                let type2 = List.find (fun (name, _) -> name = fieldName) fields2 |> snd
                try
                    (fieldName, Some (computeLUB type1 type2))
                with ex ->
                    (fieldName, None)
            )
```

For function types, we handle contravariance of arguments and covariance of return types:

```
| TFun(args1, ret1), TFun(args2, ret2) when args1.Length = args2.Length ->
    // For function types, arguments are contravariant (GLB) and return type is covariant (LUB)
    let lubArgs =
        List.map2 (fun a1 a2 ->
            if isSubtypeOf a2 a1 then a2
            elif isSubtypeOf a1 a2 then a1
            else failwith $"Cannot compute GLB for function argument types"
        ) args1 args2
    let lubRet = computeLUB ret1 ret2
    TFun(lubArgs, lubRet)
```

We also implement `computeLUBMany` to handle multiple types:

```
/// Computes the least upper bound of a list of types
let computeLUBMany (types: Type list): Type =
    match types with
    | [] -> failwith "Cannot compute LUB of empty list of types"
    | [t] -> t
    | t :: ts -> List.fold computeLUB t ts
```

Finally, the `typeCheckPatternMatch` function specifically handles the pattern matching case:

```
/// Modified type checking for pattern matching that uses LUB
let typeCheckPatternMatch (cases: Type list): Type =
    computeLUBMany cases
```

This implementation ensures that pattern matching expressions are typed according to the LUB of all their case expressions, providing more flexibility while maintaining type safety.

6.2.2 Type Checking

In the type checking phase, we extend the Typechecker module to use our LUB implementation for pattern matching expressions. The key modification is in the handling of the Match expression case:

```
| Match(expr, cases) ->
    let dups = Util.duplicates (List.map (fun (label, _, _) -> label) cases)
    if not dups.IsEmpty then
        Error([(expr.Pos, $"duplicate case labels in pattern matching: %s{Util.formatSeq dups}")])
    else
        match (typer env expr) with
        | Ok(texpr) ->
            match (expandType env texpr.Type) with
            | TUnion(unionCases) ->
                let (unionLabels, unionTypes) = List.unzip unionCases
                let caseTyper (label, v, cont: UntypedAST): TypingResult =
                    match (List.tryFindIndex (fun l -> l = label) unionLabels) with
                    | Some(i) ->
                        let env2 = { env with Vars = env.Vars.Add(v, unionTypes.[i]) }
                        typer env2 cont
                    | None ->
                        Error([(cont.Pos, $"invalid match case: %s{label}")])
                let tconts = List.map caseTyper cases
                let errors = collectErrors tconts
                if errors.IsEmpty then
                    let typedConts = List.map getOkValue tconts
                    let contTypes: Type list = List.map (fun (c: TypedAST) -> c.Type) typedConts
                    let matchTypeResult: Type = TypeInference.computeLUBMany contTypes
                    let (caseLabels, caseVars, _) = List.unzip3 cases
                    let tcases = List.zip3 caseLabels caseVars typedConts
                    Ok {
                        Pos = node.Pos
                        Env = env
                        Type = matchTypeResult
                        Expr = Match(texpr, tcases)
                    }
                else
                    Error(errors)
```

The critical improvement is in how we determine the result type of the pattern matching expression. Instead of using only the type of the first case, we now:

1. Collect the types of all pattern matching continuations (‘contTypes‘)
2. Apply ‘TypeInference.computeLUBMany‘ to compute the least upper bound of all these types
3. Use this computed LUB as the result type of the entire pattern matching expression

We have also applied the same approach to improve type checking of if-then-else expressions:

```
| If(cond, ifT, ifF) ->
    match (typer env cond) with
    | Ok(tcond) when (isSubtypeOf env tcond.Type TBool) ->
        match ((typer env ifT), (typer env ifF)) with
        | (Ok(tifT), Ok(tifF)) ->
            try
                let resultType = TypeInference.computeLUB tifT.Type tifF.Type
                Ok { Pos = node.Pos; Env = env; Type = resultType;
                     Expr = If(tcond, tifT, tifF) }
            with ex ->
                Error([(node.Pos, $"mismatching␣'then'␣and␣'else'␣types:␣"
                               + $"%O{tifT.Type}␣and␣%O{tifF.Type}"
                               + $"␣-␣{ex.Message}")])
```

For conditionals, we compute the LUB of the 'then' and 'else' branch types, allowing for more flexible type checking while maintaining type safety. This handles cases where the branches return compatible but not identical types.

6.2.3 Testing

List the tests you added under the `tests/` directory for this `hyggec` extension/improvement, e.g.:

- `tests/typechecker/pass/028-match.hyg`
- `tests/typechecker/pass029-match(mutil-union).hyg`
- `tests/typechecker/pass/030-match-nested.hyg`
- `tests/typechecker/pass/031-match-structure.hyg`
- `tests/typechecker/pass/032-match-f function.hyg`
- `tests/typechecker/fail/023-mismatch-union.hyg`
- `tests/typechecker/fail024-match-labellack.hyg`
- `tests/typechecker/fail/025-nocommen.hyg`
- `tests/typechecker/fail/026-returnlabel.hyg`

To verify the correctness and robustness of our LUB implementation, we created a comprehensive test suite that exercises various aspects of pattern matching type inference. Below we present several representative test cases that demonstrate different capabilities of our implementation.

This test verifies the core functionality of our LUB implementation using the classic Option type pattern:

```
// LUB
type OptionalInt = union {
    Some: int;
    None: unit
};

fun maybeIncrement(o: OptionalInt): OptionalInt =
  match o with {
    Some{x} -> Some{x + 1};   //  union {Some: int}
    None{_} -> None{()}       //  union {None: unit}
  };

// Test
let x = maybeIncrement(Some{41});

// Verify
assert(match x with {
    Some{y} -> y = 42;
    None{_} -> false
})
```

In this test, the 'maybeIncrement' function pattern matches on an 'OptionalInt' union type. The function returns different union cases depending on the input pattern, demonstrating how our implementation correctly computes the LUB of 'union Some: int' and 'union None: unit' to produce the expected return type 'OptionalInt'.

This test demonstrates LUB computation for more complex union types with multiple cases:

```
// Test for LUB inference with multi-branch union types
type Result = union {
   Success: int;
   Warning: string;
   Error: string
};

fun processResult(r: Result): union {Success: int; Error: string} =
 match r with {
   Success{x} -> Success{x * 2};
   Warning{msg} -> Error{msg};
   Error{err} -> Error{err}
 };

// Test different inputs
let r1 = processResult(Success{10});

// Verify results
assert(match r1 with {
   Success{x} -> x = 20;
   Error{_} -> false
})
```

This test shows how our LUB implementation handles a three-case union type, correctly inferring a two-case result type. The return type is a union that contains only ‘Success‘ and ‘Error‘ cases, even though the input pattern includes a ‘Warning‘ case that gets mapped to an ‘Error‘. The test verifies that the type system correctly handles this mapping and computes the appropriate LUB.

This test verifies LUB computation for structure types with common fields:

```
// Test for struct LUB inference
type Point2D = struct {
   x: int;
   y: int
};

type Point3D = struct {
   x: int;
   y: int;
   z: int
};

fun incompatibleStructs(cond: bool): struct { x: int; y: int } =
  if cond then
    struct { x = 1; y = 2 }            // struct { x: int; y: int }
  else
    struct { x = 1; y = 2; z = 3 };    // struct { x: int; y: int; z: int }
                                      // LUB struct { x: int; y: int }

// test
let p1 = incompatibleStructs(true);
let p2 = incompatibleStructs(false);

// result
assert(p1.x = 1);
assert(p1.y = 2);
assert(p2.x = 1);
assert(p2.y = 2)
```

This test demonstrates our implementation’s ability to compute the LUB of structure types. According to our LUB definition, the LUB of two struct types contains only their common fields with compatible types. Here, the function returns either a ‘Point2D‘ or a ‘Point3D‘, and our implementation correctly determines that the LUB is a struct containing only the common ‘x‘ and ‘y‘ fields.

This test verifies LUB computation in the context of nested pattern matching:

```
// Test for LUB inference with nested pattern matching
type Option = union {
   Some: int;
   None: unit
};

type Result = union {
   Ok: Option;
   Fail: string
};

fun processNested(r: Result): union {Some: int; None: unit; Fail: string} =
 match r with {
   Ok{opt} -> match opt with {
                Some{x} -> Some{x + 1};
                None{_} -> None{()}
```

```
               };
    Fail{msg} -> Fail{msg}
  };

// Test different inputs
let r1 = processNested(Ok{Some{41}});

// Verify results
assert(match r1 with {
    Some{x} -> x = 42;
    None{_} -> false;
    Fail{_} -> false
})
```

This test demonstrates our implementation's ability to handle nested pattern matching with multiple union types. The inner match expression returns either 'Some' or 'None', while the outer match may return 'Fail'. Our LUB implementation correctly computes the overall result type as a union of all three possible cases.

To ensure that our implementation correctly detects type mismatches, we also test error cases:

```
// LUB with type mismatch
type OptionalInt = union {
    Some: int;
    None: unit
};

fun maybeIncrement(o: OptionalInt): OptionalInt =
  match o with {
    Some{x} -> Some{true};    //  union {Some: bool} - this should cause an error!
    None{_} -> None{()}       //  union {None: unit}
  };
```

This test intentionally introduces a type error by returning a boolean in the 'Some' case where an integer is expected. Our implementation correctly detects this mismatch and reports an appropriate type error, demonstrating that our enhancement maintains the type safety of the language.

7 Partial Evaluation Optimization (Medium - Hard) — by Li, Junrui

7.1 Design

Given a typechecked AST node e, the partial evaluation process attempts to:

1. Reduce e into e' using the interpreter's reduction function
2. If reduction is successful, recursively optimize e'
3. If e cannot be reduced:
   a) If e is a simple value, return e
   b) Otherwise, recursively optimize all subexpressions of e

To ensure correct semantics, the optimization will use a specialized runtime environment where:

- Input/output functions (Reader/Printer) are set to None to prevent reduction of I/O operations
- An empty heap is used to track Pointer instances
- A constant environment tracks known constant values for variables

7.2 Implementation

I implement the partial evaluation optimization a saperate file `PartialEval.fs`, leveraging the existing *reduce()* function from `interpreter.fs`.

To meet the requirement that: the user could choose whether use this optimization or not, I set an optimization level 2 which could be enabled with argument *-O 2*.

It is notable here that, in my current mechanism, the optimization level 2 would not trigger the level 1 which is the predefined peephole optimization.

7.2.1 Constant Folding

The core implementation is in the `tryConstantFold()` function:

```
and tryConstantFold (node: AST.Node<'E,'T>) : Option<AST.Node<'E,'T>> =
    if Interpreter.isValue node then None
    else
        match Interpreter.reduce env node with
        | Some(_, reduced) ->
            match tryConstantFold reduced with
            | Some furtherReduced -> Some furtherReduced
            | None -> Some reduced
        | None -> None
```

This function recursively applies the interpreter's 'reduce' method, which already handles arithmetic operations like addition and multiplication.

The optimization also handles specific expression patterns through direct pattern matching on the `Expr` field:

```
match (lhs'.Expr, rhs'.Expr) with
| (IntVal(v1), IntVal(v2)) -> {node with Expr = IntVal(v1 + v2)}
| (FloatVal(v1), FloatVal(v2)) -> {node with Expr = FloatVal(v1 + v2)}
| _ -> {node with Expr = Add(lhs', rhs')}
```

7.2.2 Constant Propagation

For constant propagation, the optimizer maintains a mapping of variable names to their constant values using the `ConstEnv` type:

```
type ConstEnv<'E,'T> = Map<string, Node<'E,'T>>
```

This map is passed through recursive calls to `optimizeNode()` and updated when processing variable bindings. When evaluating variable references, the optimizer looks up the name in the environment:

```
| Var(name) when constEnv.ContainsKey(name) ->
    constEnv.[name]
```

For `let` bindings, the optimizer updates the environment when a variable is bound to a constant:

```
| Let(name, init, body) ->
    let init' = optimizeNode constEnv init

    let bodyEnv =
        if isConstant init' then
            constEnv.Add(name, init')
        else
            constEnv

    let body' = optimizeNode bodyEnv body
```

The `isConstant` helper function identifies AST nodes that represent constant values:

```
let isConstant (node: Node<'E,'T>) : bool =
    match node.Expr with
    | UnitVal | BoolVal(_) | IntVal(_) | FloatVal(_) | StringVal(_) -> true
    | _ -> false
```

7.2.3 Dead Code Elimination

The optimizer examines the condition after optimization and, if it's a constant, eliminates the unreachable branch:

```
| If(cond, ifTrue, ifFalse) ->
    let cond' = optimizeNode constEnv cond

    match cond'.Expr with
    | BoolVal(true) -> optimizeNode constEnv ifTrue
    | BoolVal(false) -> optimizeNode constEnv ifFalse
    | _ ->
        let ifTrue' = optimizeNode constEnv ifTrue
        let ifFalse' = optimizeNode constEnv ifFalse
        {node with Expr = If(cond', ifTrue', ifFalse')}
```

The optimizer also simplifies sequence expressions (`Seq`) by removing non-terminal effects that have no impact on the final result:

```
| Seq(nodes) ->
    let nodes' = List.map (optimizeNode constEnv) nodes
    match nodes' with
    | [] -> {node with Expr = UnitVal}
    | [last] when isConstant last -> last
    | _ -> {node with Expr = Seq(nodes')}
```

7.2.4 Function Inlining

Function inlining is implemented using pattern matching to detect these cases:

```
| Application(expr, args) ->
    let expr' = optimizeNode constEnv expr
    let args' = List.map (optimizeNode constEnv) args

    match expr'.Expr with
    | Lambda(lamArgs, body) when List.forall isConstant args' &&
                                 args'.Length = lamArgs.Length ->
        let (lamArgNames, _) = List.unzip lamArgs
        let lamArgNamesValues = List.zip lamArgNames args'
        let folder acc (var, sub) = (ASTUtil.subst acc var sub)
        let inlinedBody = List.fold folder body lamArgNamesValues
        optimizeNode constEnv inlinedBody
    | _ ->
        {node with Expr = Application(expr', args')}
```

The inlining process extracts parameter names from `lamArgs`, pairs them with argument values in `lamArgNamesValues`, and uses a `folder()` function with `ASTUtil.subst` to perform the substitution. The `List.fold` operation sequentially applies these substitutions, building the inlined function body.

7.2.5 Handling Special Cases

Avoiding Excessive Reductions To prevent reducing expressions with side effects (such as I/O operations), the implementation creates a special runtime environment where I/O operations cannot be performed:

```
let env = {
    ...
    Interpreter.Reader = None
    Interpreter.Printer = None
    ...
}
```

By setting `Reader` and `Printer` to `None`, expressions like `print("Hello")` will not be reduced and will be preserved in the optimized code.

Managing Pointer References The optimizer must handle AST nodes containing `Pointer` expressions, which are used by the interpreter but cannot be directly compiled. The `containsPointer` function recursively checks for pointer references:

```
let rec containsPointer (node: Node<'E,'T>) : bool =
    match node.Expr with
    | Pointer(_) -> true
    | Add(lhs, rhs) -> containsPointer lhs || containsPointer rhs
    // Other expression types...
    | _ -> false
```

For expressions that might create pointers, the optimizer tracks all reduction steps in a history list:

```
let history = reduceSteps node [node]
```

The optimizer then selects the most recent reduction that doesn't contain pointers:

```
match history |> List.tryFind (fun n -> not (containsPointer n)) with
| Some node -> Some node  // Use this safe reduction
| None -> None  // No pointer-free reduction found
```

7.3 Evaluation

Example command: `./hyggec rars -v -O 2 examples/helloworld.hyg`

Here the Table 1 compares the number of RISC-V assembly instructions before and after apply the partial evaluation optimization. The test cases mentioned are at folder `/project_dir/examples/`

Filename	**before**	**after**
constant_propagation.hyg	43	31
constant_folding_1.hyg	31	23
constant_folding_2.hyg	43	25
deadcode.hyg	32	21
deadcode_2.hyg	53	22
inlining.hyg	100	21
pointer.hyg	483	475

Table 1: Comparison of instruction counts before and after optimization

For the first 6 files, the optimization works well and eliminate the number of instructions of different extent since the complexity of the source code varies.

`pointer.hyg` is an example from test files of ”structures”, involing a lot of pointer expressions. The optimizer works well, it keeps most of the instructions the same.