Chapter 10: Type System (Hindley-Milner)

이 장에서는 FunLang에 **Hindley-Milner 타입 추론 시스템**을 구현한다. 명시적 타입 주석 없이도 모든 표현식의 타입을 자동으로 추론하여, 실행 전에 타입 오류를 발견할 수 있다.

개요

타입 시스템은 다음 기능을 제공한다:

- **타입 추론**: 명시적 주석 없이 타입 자동 유추
- **다형성**: `let id = fun x -> x`가 `forall 'a. 'a -> 'a` 타입을 갖음
- **정적 검사**: 실행 전에 `1 + true` 같은 오류 발견
- **CLI 통합**: `--emit-type` 플래그로 타입 표시

핵심 아이디어: Algorithm W를 사용하여 AST를 순회하며 타입 변수와 제약을 수집하고, 단일화(unification)로 해를 구한다.

구현 전략

타입 시스템은 4단계로 구현된다:

1. **Type.fs**: 타입 AST, Scheme, Substitution 연산
2. **Unify.fs**: 단일화 알고리즘과 occurs check
3. **Infer.fs**: Algorithm W 타입 추론
4. **TypeCheck.fs**: Prelude 타입 정의와 CLI 통합

왜 Hindley-Milner인가?

Hindley-Milner는 ML 계열 언어(OCaml, F#, Haskell)의 기반이다:

- **완전한 추론**: 타입 주석 없이도 항상 가장 일반적인 타입을 찾음
- **Let-polymorphism**: `let` 바인딩에서 다형성 지원
- **결정 가능성**: 추론이 항상 종료되고 결과가 유일함

Type.fs: 타입 정의

FunLang/Type.fs는 타입 시스템의 기초를 정의한다.

타입 AST

```
type Type =
    | TInt                      // int
    | TBool                     // bool
    | TString                   // string
    | TVar of int               // 타입 변수 'a, 'b, ... (정수로 구분)
    | TArrow of Type * Type     // 함수 타입 'a -> 'b
    | TTuple of Type list       // 튜플 타입 'a * 'b
    | TList of Type             // 리스트 타입 'a list
```

설계 결정: - `TVar of int`: 문자열 대신 정수 사용. 비교/해싱이 빠르고 fresh 변수 생성이 단순 - 7개 타입 생성자로 FunLang의 모든 값 표현

Scheme: 다형성 지원

```
type Scheme = Scheme of vars: int list * ty: Type
```

Scheme은 forall 변수 리스트와 타입을 포함한다: - id: forall 'a. 'a -> 'a → Scheme([0], TArrow(TVar 0, TVar 0)) - const: forall 'a 'b. 'a -> 'b -> 'a → Scheme([0; 1], TArrow(TVar 0, TArrow(TVar 1, TVar 0)))

타입 환경과 대체

```
type TypeEnv = Map<string, Scheme>
type Subst = Map<int, Type>
```

- **TypeEnv**: 변수 이름을 Scheme에 매핑
- **Subst**: 타입 변수를 구체 타입에 매핑

formatType: 타입 출력

```
let rec formatType = function
    | TInt -> "int"
    | TBool -> "bool"
    | TString -> "string"
    | TVar n -> sprintf "'%c" (char (97 + n % 26))  // 'a, 'b, ...
    | TArrow (t1, t2) ->
        let left = match t1 with TArrow _ -> sprintf "(%s)" (formatType t1) | _ -> formatType t1
        sprintf "%s -> %s" left (formatType t2)
    | TTuple ts -> ts |> List.map formatType |> String.concat " * "
    | TList t -> sprintf "%s list" (formatType t)
```

Arrow 타입의 왼쪽 피연산자만 괄호로 감싼다 (오른쪽 결합).

Substitution 연산

apply: 타입에 대체 적용

```
let rec apply (s: Subst) = function
    | TInt -> TInt
    | TBool -> TBool
    | TString -> TString
    | TVar n ->
        match Map.tryFind n s with
        | Some t -> apply s t  // 재귀! transitive chain 처리
        | None -> TVar n
    | TArrow (t1, t2) -> TArrow (apply s t1, apply s t2)
    | TTuple ts -> TTuple (List.map (apply s) ts)
    | TList t -> TList (apply s t)
```

중요: TVar 케이스에서 apply s t를 재귀 호출한다. {0→TVar 1, 1→TInt}를 TVar 0에 적용하면 TInt가 된다.

compose: 대체 합성

```
let compose (s2: Subst) (s1: Subst): Subst =
    let s1' = Map.map (fun _ t -> apply s2 t) s1
    Map.fold (fun acc k v -> Map.add k v acc) s1' s2
```

`compose s2 s1`은 "s1 먼저, 그 다음 s2" 순서다.

Free Variables

```
let rec freeVars = function
    | TInt | TBool | TString -> Set.empty
    | TVar n -> Set.singleton n
    | TArrow (t1, t2) -> Set.union (freeVars t1) (freeVars t2)
    | TTuple ts -> ts |> List.map freeVars |> Set.unionMany
    | TList t -> freeVars t

let freeVarsScheme (Scheme (vars, ty)) =
    Set.difference (freeVars ty) (Set.ofList vars)

let freeVarsEnv (env: TypeEnv) =
    env |> Map.values |> Seq.map freeVarsScheme |> Set.unionMany
```

Scheme의 free variable은 bound variable을 제외한다.

Unify.fs: 단일화 알고리즘

FunLang/`Unify.fs`는 Robinson의 단일화 알고리즘을 구현한다.

Occurs Check

```
exception TypeError of string

let occurs (v: int) (t: Type): bool =
    Set.contains v (freeVars t)
```

Occurs check는 무한 타입을 방지한다. `'a = 'a -> int`는 불가능하다.

unify 함수

```
let rec unify (t1: Type) (t2: Type): Subst =
    match t1, t2 with
    | TInt, TInt -> empty
    | TBool, TBool -> empty
    | TString, TString -> empty

    // 대칭 패턴: TVar는 양쪽 모두 처리
    | TVar n, t | t, TVar n ->
        if t = TVar n then empty
        elif occurs n t then
            raise (TypeError (sprintf "Infinite type: %s = %s"
```

```
                (formatType (TVar n)) (formatType t)))
        else
            singleton n t

    // Arrow: 도메인 단일화 → 결과 적용 → 치역 단일화
    | TArrow (a1, b1), TArrow (a2, b2) ->
        let s1 = unify a1 a2
        let s2 = unify (apply s1 b1) (apply s1 b2)
        compose s2 s1

    | TTuple ts1, TTuple ts2 when List.length ts1 = List.length ts2 ->
        List.fold2 (fun s t1 t2 ->
            let s' = unify (apply s t1) (apply s t2)
            compose s' s
        ) empty ts1 ts2

    | TList t1, TList t2 ->
        unify t1 t2

    | _ ->
        raise (TypeError (sprintf "Cannot unify %s with %s"
            (formatType t1) (formatType t2)))
```

핵심 포인트:

- `| TVar n, t | t, TVar n ->`: 대칭 패턴으로 양방향 처리
- Arrow 단일화에서 `apply s1 b1`: substitution threading
- Tuple은 길이가 같아야 하고 각 요소를 순차 단일화

Infer.fs: Algorithm W

FunLang/Infer.fs는 완전한 타입 추론을 구현한다.

freshVar: 새 타입 변수 생성

```
let freshVar =
    let counter = ref 1000  // 0-999는 Prelude용 예약
    fun () ->
        let n = !counter
        counter := n + 1
        TVar n
```

중요: 1000부터 시작하여 Prelude scheme의 bound variable(0-9)과 충돌 방지.

instantiate: Scheme 인스턴스화

```
let instantiate (Scheme (vars, ty)): Type =
    match vars with
    | [] -> ty  // Monomorphic - 그대로 반환
    | _ ->
        let freshVars = List.map (fun _ -> freshVar()) vars
```

```
        let subst = List.zip vars freshVars |> Map.ofList
        apply subst ty
```

forall 'a. 'a -> 'a를 인스턴스화하면 'x -> 'x (fresh 'x).

generalize: 타입 일반화

```
let generalize (env: TypeEnv) (ty: Type): Scheme =
    let envFree = freeVarsEnv env
    let tyFree = freeVars ty
    let vars = Set.difference tyFree envFree |> Set.toList
    Scheme (vars, ty)
```

환경에 없는 free variable을 forall로 감싼다.

infer 함수: Algorithm W 핵심

```
let rec infer (env: TypeEnv) (expr: Expr): Subst * Type =
    match expr with
    // Literals
    | Number _ -> (empty, TInt)
    | Bool _ -> (empty, TBool)
    | String _ -> (empty, TString)

    // Variable
    | Var name ->
        match Map.tryFind name env with
        | Some scheme -> (empty, instantiate scheme)
        | None -> raise (TypeError (sprintf "Unbound variable: %s" name))

    // Lambda
    | Lambda (param, body) ->
        let paramTy = freshVar()
        let bodyEnv = Map.add param (Scheme ([], paramTy)) env
        let s, bodyTy = infer bodyEnv body
        (s, TArrow (apply s paramTy, bodyTy))

    // Application
    | App (func, arg) ->
        let s1, funcTy = infer env func
        let s2, argTy = infer (applyEnv s1 env) arg
        let resultTy = freshVar()
        let s3 = unify (apply s2 funcTy) (TArrow (argTy, resultTy))
        (compose s3 (compose s2 s1), apply s3 resultTy)

    // Let with polymorphism
    | Let (name, value, body) ->
        let s1, valueTy = infer env value
        let env' = applyEnv s1 env
        let scheme = generalize env' (apply s1 valueTy)
        let bodyEnv = Map.add name scheme env'
```

```
        let s2, bodyTy = infer bodyEnv body
        (compose s2 s1, bodyTy)

    // ... If, LetRec, Tuple, List, Match, etc.
```

핵심 패턴:

1. infer는 (Subst, Type) 튜플 반환
2. 매 단계에서 이전 substitution을 환경에 적용: applyEnv s1 env
3. Let에서 generalize 호출 → let-polymorphism
4. Lambda param은 Scheme([], ty) → monomorphic

Let-Polymorphism vs Lambda Monomorphism

Let은 다형성을 허용:

```
let id = fun x -> x in (id 5, id true)
// id: forall 'a. 'a -> 'a
// 첫 번째 사용: 'a = int
// 두 번째 사용: 'a = bool
// 결과: (int, bool) ✓
```

Lambda parameter는 monomorphic:

```
fun f -> (f 1, f true)
// f는 한 타입으로 고정
// f: int -> 'a 또는 f: bool -> 'a 둘 중 하나만 가능
// 타입 오류!
```

TypeCheck.fs: 통합

FunLang/TypeCheck.fs는 Prelude 타입과 typecheck 함수를 제공한다.

initialTypeEnv: Prelude 함수 타입

```
let initialTypeEnv: TypeEnv =
    Map.ofList [
        // map: ('a -> 'b) -> 'a list -> 'b list
        "map", Scheme([0; 1], TArrow(TArrow(TVar 0, TVar 1), TArrow(TList(TVar 0), TList(TVar 1))))

        // filter: ('a -> bool) -> 'a list -> 'a list
        "filter", Scheme([0], TArrow(TArrow(TVar 0, TBool), TArrow(TList(TVar 0), TList(TVar 0))))

        // fold: ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
        "fold", Scheme([0; 1], TArrow(TArrow(TVar 1, TArrow(TVar 0, TVar 1)), TArrow(TVar 1, TArrow(TList(TV

        // id: 'a -> 'a
        "id", Scheme([0], TArrow(TVar 0, TVar 0))

        // ... length, reverse, append, const, compose, hd, tl
    ]
```

typecheck 함수

```
let typecheck (expr: Expr): Result<Type, string> =
    try
        let subst, ty = infer initialTypeEnv expr
        Ok(apply subst ty)
    with
    | TypeError msg -> Error(msg)
```

CLI 통합

Program.fs에서 --emit-type 플래그 처리:

```
| EmitType ->
    let source = results.GetResult Expr
    let ast = parse source
    match typecheck ast with
    | Ok ty -> printfn "%s" (formatType ty)
    | Error msg -> eprintfn "Error: %s" msg; exit 1
```

Examples

리터럴 타입 추론

```
$ dotnet run --project FunLang -- --emit-type -e '42'
int

$ dotnet run --project FunLang -- --emit-type -e 'true'
bool

$ dotnet run --project FunLang -- --emit-type -e '"hello"'
string
```

함수 타입 추론

```
$ dotnet run --project FunLang -- --emit-type -e 'fun x -> x'
'm -> 'm

$ dotnet run --project FunLang -- --emit-type -e 'fun x -> x + 1'
int -> int

$ dotnet run --project FunLang -- --emit-type -e 'fun x -> fun y -> x + y'
int -> int -> int
```

Let-Polymorphism

```
$ dotnet run --project FunLang -- --emit-type -e 'let id = fun x -> x in (id 5, id true)'
int * bool
```

```
$ dotnet run --project FunLang -- --emit-type -e 'let id = fun x -> x in id'
'm -> 'm
```

리스트와 튜플

```
$ dotnet run --project FunLang -- --emit-type -e '[1, 2, 3]'
int list

$ dotnet run --project FunLang -- --emit-type -e '[]'
'm list

$ dotnet run --project FunLang -- --emit-type -e '(1, true, "hello")'
int * bool * string
```

Prelude 함수 타입

```
$ dotnet run --project FunLang -- --emit-type -e 'map'
('m -> 'n) -> 'm list -> 'n list

$ dotnet run --project FunLang -- --emit-type -e 'filter'
('m -> bool) -> 'm list -> 'm list

$ dotnet run --project FunLang -- --emit-type -e 'map (fun x -> x + 1)'
int list -> int list
```

타입 오류

```
$ dotnet run --project FunLang -- --emit-type -e '1 + true'
Error: Cannot unify int with bool

$ dotnet run --project FunLang -- --emit-type -e 'if true then 1 else false'
Error: Cannot unify int with bool

$ dotnet run --project FunLang -- --emit-type -e 'let rec f x = f in f'
Error: Infinite type: 'm = 'm -> 'n
```

재귀 함수 타입

```
$ dotnet run --project FunLang -- --emit-type -e 'let rec fact n = if n <= 1 then 1 else n * fact (n - 1) in
int -> int
```

구현 세부 사항

Substitution Threading

매 추론 단계에서 이전 substitution을 다음 단계에 전파해야 한다:

```
| App (func, arg) ->
    let s1, funcTy = infer env func
    let s2, argTy = infer (applyEnv s1 env) arg  // ← s1 적용
    let s3 = unify (apply s2 funcTy) (TArrow (argTy, resultTy))
    (compose s3 (compose s2 s1), apply s3 resultTy)
```

Type Variable 충돌 방지

Prelude scheme은 0-9 범위의 bound variable을 사용하고, freshVar는 1000부터 시작한다:

```
// Prelude: Scheme([0; 1], TArrow(TVar 0, TVar 1))
// Fresh: TVar 1000, TVar 1001, ...
```

Generalize 타이밍

Let에서 value 추론 후, substitution을 환경에 적용한 다음 generalize해야 한다:

```
| Let (name, value, body) ->
    let s1, valueTy = infer env value
    let env' = applyEnv s1 env          // 먼저 적용
    let scheme = generalize env' (apply s1 valueTy)  // 그 다음 generalize
```

정리

이 장에서 구현한 내용:

기능	파일	설명
타입 AST	`Type.fs`	7개 타입 생성자 (TInt, TBool, TString, TVar, TArrow, TTuple, TList)
Substitution	`Type.fs`	apply, compose, freeVars
단일화	`Unify.fs`	occurs check, unify
타입 추론	`Infer.fs`	Algorithm W (freshVar, instantiate, generalize, infer)
통합	`TypeCheck.fs`	initialTypeEnv, typecheck
CLI	`Program.fs`	–emit-type 플래그

Hindley-Milner의 특징:

특성	설명
완전한 추론	타입 주석 없이 가장 일반적인 타입 유추
Let-polymorphism	let 바인딩에서 다형성 지원
결정 가능성	항상 종료, 유일한 principal type

특성	설명
Occurs check	무한 타입 방지

테스트

```
# fslit 통합 테스트
make -C tests type-inference
make -C tests type-errors

# Expecto 단위 테스트
dotnet run --project FunLang.Tests

# 전체 테스트 (460개)
make -C tests && dotnet run --project FunLang.Tests
```

소스 참조

전체 소스 코드는 다음 위치에서 확인할 수 있다:

- **FunLang/Type.fs**: 타입 정의, Substitution, Free Variables
- **FunLang/Unify.fs**: 단일화 알고리즘, Occurs Check
- **FunLang/Infer.fs**: Algorithm W 구현
- **FunLang/TypeCheck.fs**: Prelude 타입, typecheck 함수
- **FunLang/Program.fs**: CLI 통합 (–emit-type)

관련 문서

- implement-hindley-milner-algorithm-w - Algorithm W 전체 흐름
- implement-let-polymorphism - Let-polymorphism 구현
- avoid-type-variable-collision - 타입 변수 충돌 방지
- write-type-inference-tests - 타입 추론 테스트 방법