The Design and Implementation of Typed Scheme

Sam Tobin-Hochstadt & Matthias Felleisen

Why: From Scripts to Programs

- Values can freely flow back and forth between typed and untyped modules
- Integrate type checker with the macro expander
- Combine the idea of occurrence typing with subtyping, recursive types, polymorphism and inference

Overview of Typed Scheme

- True union types
- First-class polymorphic functions
- Specify recursive types, as well as constructors and accessors that manage them
- Base types to match those of Racket

A Formal Model

- *Figure 1* serves only as λ_{TS} of occurrence typing
- Expressions: value, application, conditional or variable
- Values: abstraction, number, boolean or constant
- Types: T, function, base, union, collection

Typing Rules

- Visible predicates accumulate information about expressions
- Latent predicates accommodate programmer-defined functions that are used as predicates
- Supports logical combinations of predicates
- Meaning: $\Gamma \vdash$ e : τ ; ψ

Rules to Note

- T-Abs vs T-AbsPred
    - Gives an abstraction a latent predicate
- T-App vs T-AppPred
    - Produces **true** if and only if *x* has a value of type σ
- *Auxiliary* operations and *Environment* operations

Proof-Theoretic Typing Rules

- What happens if #*f* is passed in to previous example?
- Type soundness is introduced in *Figure 6*

From λ_{TS} to Typed Scheme

- Parametric polymorphism
- Type inference
    - *let**, *letrec*
    - Type arguments to polymorphic functions

Adapting Scheme Features

- *define-struct* is *the* fundamental method for constructing new data types
  - This supports recursive types as well as extensions
- Variable-arity, multiple-return values and *apply*
- *filter : (All (a b) ((a -> Boolean) (Listof a) -> (Listof b))*
- *call/cc : (All (a) (((a -> ⊥) -> a) -> a))*

Programming in the Large

- Racket has a first-order module system
- Typed Scheme requires dynamic checks at the module boundary with *require/typed*
- Handling macros with *local-expand* primitive

Related Work

- Soft typing: type inference to assist debugging programs statically
  - Programmers should not have to write down type definitions
- Hindley-Milner vs Shiver & Aiken and Heintze
- Gradual typing: integrate typed and untyped programs

Follow-ups

- This covered a lot of implementation “with” TS, not “of” TS
- TAPL
    - Sets, Relations, and Functions (2.1)
    - Safety = Progress + Preservation (8.3)
    - Intersection and Union Types (15.7)