[witty title]: Towards a Practical Type System [hopefully replace with programming language] for the Enriched Effects Calculus.

Abstract

Introduction

Motivation

- Motivation for doing the project, reworked from present Literature and Technology Survey.

Goals

- Create usable syntax
- User can define and type check terms.
- Provide a practical environment for iterative development.

Literature and Technology Survey

- Introduction to computational effects
- Algebraic effects
- Monads and transformer stacks
- Type theory
- Linear types
- lambda calculus
- Simply-typed lambda calculus
- Enriched effects calculus
  - Relation to monads
  - comparison to linear Haskell

Development

Tools Used

- Dotty Scala 0.14.0.
- ANTLR 4.
- PPrint from Li Haoyi.

Requirements Specification

Functional Requirements

Non-Functional Requirements

- Extensible

    - Try to limit special cases.
    - Do not expose Internal APIs.
- etc.

Design and Architecture

- Modular design, separate compiler library to REPL
- Compiler
    - Parser
    - Namer
    - Typer
    - Context
- REPL
    - Environment printing.
    - AST printing.
    - define terms
    - type check terms/source files
    - Good error messages
    - Erase from scope if no Type

Iteration of development

- Initially basic effect calculus
- Add Stoup to EC
- Introduce linear types.
- Introduce data declarations.

Forming a Usable Syntax

- Based from Scala grammar, adapted to Haskell style and addition of EEC terms.

Comparison of syntax to EEC

- User defined value sum types should be isomorphic to EC+ sums
- User defined computational sum types should be isomorphic to EEC sums.
- User defined types currently must have more than one constructor, so `()` remains a terminal object, and `Void` and `Void#` are the sole types with no members.
- n-ary products isomorphic to nested binary products.
- `fst` and `snd` primitives achieved with match expressions.
    - need to show how linearity is preserved

Syntactic Sugar and Reduction

- arbitrary depth patterns to single depth
- patterns in `let` expressions

- infix application
- `if` to case expressions on Booleans
- top level statements map to variable references
- Literals map to 0-ary function symbols.

Type checking

- Syntax requires checking of declared types.
- Environment contains three maps
  - names to type declarations
  - names to variable declarations
  - names to linear variable declarations
- Parametric polymorphism with sub typing to constrain to computations only.
  - Unification of types, no explicit type parameter syntax.
  - If at point of declaration, no type exists of that name, it is considered a type variable.
- Every time a term with type variables is referenced, unique type variables are generated and then unified.
- Unification algorithm.
- Addition of Stoup constraints required new term introductions.
  - should function declarations be considered as lambdas, when referenced which allows the body to have linear dependencies, or as function symbols, which must have no linear dependencies? - this impacts the necessity of introducing `?`.
- Compare with Hindley-Milner unification.

Pattern matching

- generation of templates based on lookup of the type
- special cases for primitives such as Boolean and products.
- comparison to other systems e.g. Liu (2016)
- why do patterns not exist for `!A` and `!A *: B#` types?
  - implicit non termination case.
- as of current, requires type checking.

Pattern Matching Algorithm

- Look at scrutiny type, e.g. `case e of ...` where `e: Either (A, B) C`
- Build a stack of template patterns required to prove the scrutiny type:
  - Lookup constructor signatures that are mapped to the scrutinee type.
    - The constructor for any base type is a wildcard sentinel.
  - For each constructor signature, push to function-stack a function that can construct a template for a pattern from the template-stack and push it back to the template-stack.
  - The wildcard sentinel will push an `Ident` template.

    - Unify the types of the constructor arguments with the current scrutinee type. Recurse on the type arguments.
    - Fold over the function-stack to build a stack of final templates.
- Iterate through case clauses, each time a pattern unifies with the top of the stack, pop it off. When a pattern fails to unify with the top of the stack, continue to the next case clause.
    - An `Ident` pattern unifies with any template.
    - A `Literal` pattern unifies with no template.
    - All other patterns unify with a template of the same shape.
- After all cases, a match expression that is total with respect to matching implies an empty stack.

Testing

- Declarative test framework to check:
    - Expressions that should type check.
    - Expressions that are syntactically valid but have no type.
    - Collections of top level definitions in the same context.

Examples and Evaluation

- Isomorphisms from Proposition 4.1 in Egger, Ejlers and Simpson (2014).
- Pick good examples from testing, e.g. why `\(f: A -> B#) =>. \(x: !A) => let !y = x in f y is bad`
- show that you can only shadow linear var after it is consumed.
- Create some encodings of effects and sample programs to use them.

Missed Goals

- Different precedence operators.
- at present - any normalising interpreter.

Conclusions

References

Egger, J., Ejlers, R. and Simpson, A., 2014. The enriched effect calculus: syntax and semantics. Journal of Logic and Computation, 24(3), pp.615-654.

Liu, F., 2016, October. A generic algorithm for checking exhaustivity of pattern matching (short paper). In Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (pp. 61-64). ACM.