Seminar: Programming Language Features

Marco Tzschentke, Prof. Dr. Klaus Ostermann

University of Tübingen

Course Information

- We meet every week on Thursday, 12-14h, in C215
- Course Material and information can be found on `https://ps.cs.uni-tuebingen.de/teaching/ss26/pl/`
- Seminar for Master students, but also possible for Bachelor students
- 3 ETCS

Course Overview

- Each student will get one of the topics (see following slides) assigned
- Every week, one student will do a presentation about their topic
- 35 minutes presentation, afterwards 10 minutes discussion
- Each student will also prepare a term paper about their topic (until the end of semester)

Registration

- To register for the course, please write an email to `marco.tzschentke@uni-tuebingen.de`
- Deadline for registrations is the 23.04.2026
- Your email should contain the following information
  - matriculation number
  - name
  - course of study + degree
  - number of your current semester
  - your student e-mail address
  - Your chosen topic(s)

Presentation

For your presentation you will have to

- Read the relevant resources
- Potentially collect additional resources
- Prepare a 35 minute presentation
- Briefly discuss your presentation with me before holding it

AI usage

- You can use AI to help with understanding concepts / formulate sentences etc
- Don't rely on AI, read and understand everything yourself
- If you use AI to generate parts of your presentation (e.g. images), say so and check for correctness

Topics

- From the following topics, please choose 1-3 you are interested in
- Please provide these in your registration email
- If multiple people pick the same topic, the topic will either be split, or someone will have to take their second choice
- If you are unsure about the difficulty etc, feel free to ask, we can usually pick an easier or harder topic instead

Parametric Polymorphism and Hindley-Milner Type Inference

- A simple type system (e.g. in simply typed lambda calculus) has atomic types ($\texttt{Bool}, \texttt{Int}, \ldots$) and composite types ($\texttt{Bool} \to \texttt{Int}, \texttt{List[Int]}, \ldots$)
- In this case, terms allowing *arbitrary* types, e.g. $\lambda x.x$ (identity function) cannot be typed in such a system, or require annotations, e.g. $\lambda x : \texttt{Int}.x$
- To solve this, HM Type Systems contain *Type Variables* and allow types such as $\forall X.X \to X$
- Milner's Algorithm W can be used to simply infer types in such a system (without requiring annotations)

Algebraic Subtyping

- Type System with subtyping commonly have issues ensuring algebraic properties still hold
- *Extensibility*: Existing programs should remain well-typed when new types are added
- *Principal Types*: Every Term/Program should have one most general type (that can be inferred)
- Usually, Extensibility and Principal Types are ensured *after* syntactially defining types and their relationships
- Dolan's Algebraic Subtyping approach instead defines an abstract type lattice *first*, which then ensures the type system has the desired properties

Generalized Algebraic Data Types

- *Algebraic Data Types* (PI 1): Combination of unions and products with recursion, e.g. lists, binary trees, sums, ...
- *Generalized* ADTs: Encode more information about constructors/arguments in their types

```
data Expr a where
  Lit  :: Int -> Expr Int
  Bool :: Bool -> Expr Bool
  If   :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Type Classes/Traits

- Also known as ad-hoc polymorphism, similar to interfaces in OOP
- Allows defining behaviour for different types, regardless of their implementation details
- Commonly used for arithmetic operations (sums, products, etc) or pretty printing (`Show` in Haskell)

```
class Add a where
add: a -> a -> a

instance Add Int where
add: Int -> Int -> Int
add a b = a + b

instance Add String where
add: String -> String -> String
add a b = a ++ b
```

Continuations

- Way to make the execution order of a program explicit
- Often used in compilers (CPS translation) or for error handling

```
let rec tak x y z =
if y < x then
tak
(tak (x - 1) y z)
(tak (y - 1) z x)
(tak (z - 1) x y)
else z
```

```
let rec cps_tak x y z k =
if x <= y then k(z)
else
cps_tak (x - 1) y z
(fun v1 ->
cps_tak (y - 1) z x
(fun v2 ->
cps_tak (z - 1) x y
(fun v3 ->
cps_tak v1 v2 v3 k)))
```

Algebraic Effects

- Effects, that is side effects like console output, or exceptions are tracked on the type level
- Allows functional languages to reason about possible effects while remaining functionally pure
- Generalize many common language features like I/O, exceptions, generators, continuations, ...

```
effect throw(msg: String): Nothing

def div( a: Double, b: Double): Double / throw =
if (b == 0.0) {
do throw("division by zero")
} else {
a / b
}
```

Lazy Evaluation

- Alternative to common call-by-value or call-by-name evaluation order
- Expressions are only evaluated just before their results are needed
- These results are cached and reused when the same expression appears again
- Ensures that any computation is only calculated at most once
- Difficult if a language has side effects (e.g. printing to stdout)
- Side effects are also only executed once which might lead to unexpected results

Monads

- Category Theory: A monad is a monoid in the category of endofunctors
- In functional programming, particularly Haskell, used to encode non-functional concepts
- Common examples: IO monad, state monad, reader monad, error monad, ...
- Defined by operations `return :: a -> M a` and `bind :: (M a) -> (a -> M b) -> M b`
  - `return :: a -> M a`, introduces a mondad `M`
  - `bind :: (M a) -> (a -> M b) -> M b`, "chains" monadic computations
- Allows carrying state, possible errors etc without manually having to wrap and unwrap the monadic context
- Error/Option monad can be compared to the ? operator in Rust

Array Programming

- Many functional operations `map`, `filter`, `...` operate on independent values
- Similarly, graphical, neural network, etc. operations have many similar independent calculations
- Such operations can be paralellized, massively speeding them up
- Futhark, Octave, NumPy, etc. directly allow operations on array-like structures (tables, vectors, etc.)
- When compiling/running such code, these optimizations can then directly be applied
- Used in some form for most computations on large data sets

Refinement Types

- Often times, computations have known error conditions
- Common examples: division by zero, head of empty list
- Idea: encode properties of terms within types
- Instead of `div: Int -> Int -> Int`, have `div: Int -> x:Int // x != 0 -> Int`
- ensures the second argument (divisor) is nonzero, then division by zero is impossible by the type system
- Less powerful than full dependent types, but easier to implement

Gradual Typing

- Many languages, e.g. Python, Javascript, etc have non-strict type systems
- In such languages, even during runtime type unsafe programs might "just work"
- This often leads to unexpected bugs
  "b" + "a" + +"a" + "a"; // -> 'baNaNa'
- Gradual Typing enforces *some* properties in the type system to assert things during runtime
- Most common examples are python type hints or TypeScript

Linear Types

- Logically: Remove Weakening and Contraction Rules from a system
- In terms of programming languages, this corresponds to enforcing that any value is used *exactly* once
- That is, memory will never be allocated and not be used, nor can memory be cloned
- Rust's ownership system is close to a linear (more accurately an affine) system to enforce memory safety
- Linearity enforces some desirable properties
  - File Handles/Network Connections are never dangling but have to be closed
  - Memory usage is very controlled
  - Concurrency is more easily enforced
  - ...

Logic Programming

- Programming Paradigm often used for symbolic manipulation
- Examples include: database systems, natural language processing, compiler construction, ...
- Programs are lists of *Horn Clauses* `A :- B, C, D, ...`, which can be read as $B \wedge C \wedge D \wedge \cdots \Rightarrow A$
- Given a list of Horn clauses, computations are then performed by specifying a query "Is proposition X true"
- The interpreter than uses rewriting rules to try and reduce X to some known proposition, that is a Horn clause `A :-` without assumptions

Closures

- Most programming languages have some form of higher order functions or closures
- Function Pointers in C, Lambdas in Python, Delegates in C#, Function Objects in C++, ...
- All of these differ wildly in their implementation and features
  - Partial Application
  - Capturing Environment
  - Currying/Uncurrying
  - ...

Dependent Type Systems

- Types that depend on Terms
- Many other features are special cases (Refinement Types, GADTs, etc.)
- Very commonly used in proof assistants (Curry-Howard isomorphism)
- Most simple version: Vectors with included length `Vec a n`
- Then we can have `push :: n:Nat -> Vec a n -> a -> Vec a (n+1)`