Adding a type system to an untyped language

A journey to a type save environment for developers

Christoph Bühler
OST Eastern Switzerland University of Applied Sciences
MSE Seminar “Programming Languages”
Supervisor: Farhad Mehta
Semester: Autumn/Spring 2020

Abstract

Short abstract about why a developer should understand the lambda calculus. What is is how it works and why it is relevant in software engineering and other fields.

Keywords: Lambda-Calculus, type systems, simple types

1 Introduction

All modern programming languages have type systems. They are either of a more dynamic nature - like JavaScript - or statically typed like C#. Functional languages like Haskell have an even stricter form of a type system. They all have one thing in common: They aid the developer to create programs, without the constant fear about runtime errors.

This paper shall give the reader an idea of the steps that are needed to create a type system and how it is applied to an untyped language. This paper will use JavaScript and TypeScript[1] as examples for certain comparisons. Of course, JavaScript is not an untyped language, but it does not have a strict static type analysis which can lead to runtime errors during interpretation and execution of the code. TypeScript is a superset of the JavaScript language which fills this gap and adds a static type analyzer as well as some sort of a compiler (i.e. transpiler) to the language.

But why type systems are helpful and how they work is not a trivial question to be answered. Consider the following code statements:

```
foo = "Hello World"
bar = 42
foo - bar // NaN (Not a Number)
```

As humans, we immediately understand that this statement is not going to terminate well - assuming we have an untyped language. Strings and numbers are not of the same type and cannot be added together. To determine that this is not going to work, the computer needs to execute the statements one by one and will encounter a wrong state. A type system can prevent such errors and create a human-readable message when compiling such a program.

The reader should have an understanding of programming languages and a brief understanding of the untyped λ-Calculus which is described in the first chapters of “Types and Programming Languages” by Benjamin C. Pierce [Pie02].

The following sections will describe the basics of the λ-Calculus, what “types” and “simple types” are and how they can be applied to the λ-Calculus to achieve the “simply typed λ-Calculus”.

[1] https://www.typescriptlang.org/

2 Lambda-Calculus (λ-Calculus)

A computer program can be described in various ways. One very famous variant is the “turing machine” which was defined in a journal [Tur37] by A. M. Turing in the year 1937. The turing machine is fed with instructions and contains a “memory” band to write down results for further computation.

Another famous - but more abstract - method to describe a computation is the λ-Calculus. This system was specified by Alonzo Church [Chu41]. It is a mathematical model of a computation that only contains three rules of operation [Pie02]. Those three operations can be viewed in the following “grammar”:

$$
\begin{array}{lcll}
\mathtt{t} & ::= & & \textit{terms}: \\
 & | & \mathtt{x} & \textit{variable} \\
 & | & \lambda\mathtt{x.t} & \textit{abstraction} \\
 & | & \mathtt{t\ t} & \textit{application}
\end{array}
$$

The untyped λ-Calculus is turing complete, which means it can compute *any* program and therefore can run infinitely. In an untyped λ system, it is possible to search for the successor of “true”, which requires the argument to be a number and therefore results in a stuck state.

Since it is not desirable for computer programs to run to infinity, there has to be a way to split up computer programs into two categories: The “useful” and “useless” ones. Any program that will run forever or that will be stuck in an error state counts towards the “useless” ones. Other programs that have valid input and output will be counted to the “useful” programs.

TODO: reference to simons paper about the untyped calculus

For the further progress of this paper, it is necessary to recall the grammar for arithmetic expressions of the untyped calculus [Pie02]:

2.1 Terms

t	::=		*terms*:
	|	true	*constant true*
	|	false	*constant false*
	|	if t then t else t	*conditional*
	|	0	*constant zero*
	|	succ t	*successor*
	|	pred t	*predecessor*
	|	iszero t	*zero test*

2.2 Values

v	::=		*values*:
	|	true	*true value*
	|	false	*false value*
	|	nv	*numeric value*

2.3 Numeric Values

nv	::=		*numeric values*:
	|	0	*zero value*
	|	succ nv	*successor value*

3 Types and Simple Types

To get one step closer towards the goal of having a simply typed λ-Calculus we need to define what a type is.

3.1 Types

A type is a classification of a value or multiple values. We typically use mathematical terms to describe a type and the relation of values to their types. When we say "$x : T$", we mean that "x is of type T", or in mathematical terms: "$x \in T$" [Pie02]. This relation makes it possible to determine the type of any computation that correlates with x and can be *statically* analyzed. For example:

$$\begin{cases} \text{if} & x, y \in \mathbb{N} \\ \text{then} & f(x, y) = x + y \in \mathbb{N} \end{cases}$$

We can determine the result type of f without *running* the function. Shown in typescript syntax this could mean:

```typescript
const x: number = 1
const y: number = 2
const f = (x, y) => x + y
f(x, y) // result is also of type number
```

With such a possibility, we can rule out stuck or meaningless programs. But how do we get there? We add the typing relation to the grammar and define, that terms and variables must have types:

t	::=		*terms*:
	|	x: T	*variable*
	|	λx: T.t	*abstraction*
	|	. . .	

To express types correctly, we also need new syntactic forms:

T	::=		*types*:
	|	Bool	*type of booleans*
	|	Nat	*type of natural numbers*

And in addition to the syntactic definition, we need typing rules that define the types of the arithmetic expression on top of the rules that are given by the untyped λ-Calculus.

$$\texttt{true : Bool}$$
$$\texttt{false : Bool}$$
$$\texttt{0 : Nat}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : \texttt{T} \quad t_3 : \texttt{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \texttt{T}}$$

$$\frac{t_1 : \texttt{Nat}}{\text{succ } t_1 : \texttt{Nat}}$$

$$\frac{t_1 : \texttt{Nat}}{\text{pred } t_1 : \texttt{Nat}}$$

$$\frac{t_1 : \texttt{Nat}}{\text{iszero } t_1 : \texttt{Bool}}$$

This means, we allocate `true` and `false` to the `Bool` type. Then we assign `0` to the `Nat` type. The derivation rules state that, all `succ` (successors) and `pred` predecessors are of the `Nat` type. The rules for the predecessor and successor define, that the result of the applied `succ` or `pred` function must be of type `Nat`. The rule for the condition defines that the input for the "`if`" must be a boolean value and the result is of type `T`. Both branches of the condition must have the same type.

3.1.1 Typesafety.

Well-typed programs cannot "go wrong."

– Robin Milner (1978)

The given typing rules give our typing system a pretty important property: *safety* (or *soundness*) [Pie02]. In conjunction with the normalization property [Pie02] [BN98], which eliminates the turing completeness in our system, we can guarantee that our programs that compile successfully with this type system won't ever go wrong. We can call such a program well-typed (i.e. it compiles according to the given typing rules [Car96]).

This *safety* is defined by two theorems [Pie02]:

- *Progress*: Well-typed terms are not stuck. They can take a step in the evaluation rules or are a value.
- *Preservation*: A well-typed term that takes a step in the evaluation rules will yield a result that is also well-typed.

3.2 Intermediate Result

With the given syntax, evaluation rules and type definitions, we would have a type system that could successfully compile the following lines of code (the syntax is inspired by TypeScript for a clear reference to a computer program):

```
const tr: boolean = true
const x: number = 42

if (tr) {
    x
} else {
    x + 1
}
```

3.3 Simple Types

Alonzo Church defined the theory of simple types [Chu40]. In combination with the examples and statements of Benjamin C. Pierce [Pie02], there exists a definition of a simple type. Simple types are a first approach towards a typesafe environment for developers. They contain “base types” (or “value types”) like `Bool` and `Nat` (natural numbers) as well as “function types”.

Function types are needed to grant the program the possibility to perform computations. Up until now, we can allocate variables to types and can perform an *if* condition.

3.3.1 Base Types. Base types represent unstructured values in a programming language [Pie02]. An incomplete list of such base types we will encounter is:

- Numbers (`Integers` and `Float`)
- Booleans
- Strings (list of `Characters`)

Since base types are unspectacular and are used to calculate other types, literature often substitutes them with a letter for all *unknown* base types [Pie02]. Often, constructs like $\mathcal{A}$ or other letters are seen. For this paper we will establish the following syntactic form from Benjamin C. Pierce [Pie02]:

$$\begin{array}{lcll} \mathsf{T} & ::= & & \textit{types}: \\ & | & \ldots & \\ & | & \mathsf{A} & \textit{base type} \end{array}$$

3.3.2 Function Types. From a theoretical perspective, this language is quite interesting, but to be used as a programming language, it lacks some needed features. For example, we need the possibility to apply a function to some input to generate some output. Otherwise this programming language will be quite boring. To add functions to our typing syntax we add the following line:

$$\begin{array}{lcll} \mathsf{T} & ::= & & \textit{types}: \\ & | & \ldots & \\ & | & \mathsf{T}{\to}\mathsf{T} & \textit{type of functions} \end{array}$$

A *type environment* (Γ) is introduced in the following derivation rules. This environment (or sometimes called *type context*) is a set with variables and their types [Pie02]. When our type-checker starts, the starting type environment equals “$\varnothing$” (the empty set). With each evaluation step, this set will grow and contain the specified values.

Now we have the typing syntax for functions, but we need some additional typing rules to ensure the types of functions can be calculated statically:

$$\frac{\Gamma, x : \mathsf{T}_1 \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \lambda x : \mathsf{T}_1 . t_2 : \mathsf{T}_1 \to \mathsf{T}_2} \qquad \text{(Abstraction)}$$

This typing rule for the general abstraction evaluation rule of the λ-Calculus TODO: ref to simons paper adds a premise to our system that translates to “if x is of type T_1 and is in our typing context Γ and the term t_2 is of type T_2, then the abstraction $\lambda x : T_1.t_2$ has the type $T_1 \to T_2$”.

Also an additional typing rule for variables and applications are needed:

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}} \qquad \text{(Variable)}$$

“The type that is assumed for x is in the set of Γ”.

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_{11} \to \mathsf{T}_{12} \quad \Gamma \vdash t_2 : \mathsf{T}_{11}}{\Gamma \vdash t_1 t_2 : \mathsf{T}_{12}} \qquad \text{(Application)}$$

“If t_1 is a function that takes a T_{11} and returns a T_{12} and the term t_2 is a value of type T_{12} then the result of the application of t_1 to t_2 will be of type T_{12}”.

Translated into a programming language:

```
// number (Variable)
const x = 1337

// number => string (Abstraction)
const f = nr => nr.toString()

// string (Application)
const r = f(x)
```

4 λ-Calculus, Simple Types and Extensions

When we apply “simple types” to the purely untyped λ-Calculus, the result is the “pure simply typed λ-Calculus”. This could count as a programming language since we can perform all basic operations a computation needs. We’re also able to analyze the code and calculate the types needed for functions and variables and therefore can categorize our programs into meaningful and meaningless ones. To make the syntax more useful however, we can extend our simple types with “simple extensions” which do not include any form of polymorphism. Those extensions make our language and the type checker more useful and able to perform operations.

The following sections will explain such extensions and how they are constructed. For the further reading, the rules of the “pure simply typed λ-Calculus” are stated again [Pie02]:

Syntax

t	::=		*terms*:
	|	x	*variable*
	|	λx: T . t	*abstraction*
	|	t t	*application*

v	::=		*values*:
	|	λx: T . t	*abstraction value*

T	::=		*types*:
	|	T→T	*type of functions*

Γ	::=		*contexts*:
	|	∅	*empty context*
	|	Γ, x: T	*term variable binding*

Evaluation

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2} \qquad \text{(Application 1)}$$

"If there exists an evaluation step from t_1 to t_1' take this step prior to the evaluation step of t_2". This forces the program, to first evaluate all terms of t_1 until the next rule hits.

$$\frac{t_2 \longrightarrow t_2'}{v_1 t_2 \longrightarrow v_1 t_2'} \qquad \text{(Application 2)}$$

"If there exists an evaluation step from t_2 to t_2' and the left side of the application is already a value, take this step". This defines that when the lefthand side of the application is reduced to a value, evaluate the righthand side.

$$(\lambda x\colon T_{11}.t_{12})v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(Application Abstraction)}$$

"Replace the variable x with the value v_2 in the term t_{12}". This represents the effective computation or application of the value to a term.

Typing

$$\frac{x\colon \mathtt{T} \in \Gamma}{\Gamma \vdash x\colon \mathtt{T}} \qquad \text{(Variable)}$$

$$\frac{\Gamma \vdash t_1\colon \mathtt{T}_{11} \longrightarrow \mathtt{T}_{12} \quad \Gamma \vdash t_2\colon \mathtt{T}_{11}}{\Gamma \vdash t_1 t_2\colon \mathtt{T}_{12}} \qquad \text{(Application)}$$

$$\frac{\Gamma, x\colon \mathtt{T}_1 \vdash t_2\colon \mathtt{T}_2}{\Gamma \vdash \lambda x\colon \mathtt{T}_1.t_2\colon \mathtt{T}_1 \longrightarrow \mathtt{T}_2} \qquad \text{(Abstraction)}$$

This in combination with base and function types will be the cornerstone of the "simple extensions" which we will see in the next sections. Those sections will introduce new elements to the given categories ("syntax", "evaluation", "typing").

4.1 Unit Type

The `unit` type represents a useful type often found in functional programming languages like Haskell or F#. It is used to "throw away" a computation result and combine multiple computations together [Pie02]. In Haskell this can be used in the `main` function to glue several functions together that contain side effects. It can be viewed as the "`void`" type in C# or Java [Pie02].

Addition to the syntax [Pie02]:

t	::=		*terms*:
	|	. . .	
	|	unit	*constant unit*

v	::=		*values*:
	|	. . .	
	|	unit	*constant unit*

T	::=		*types*:
	|	. . .	
	|	Unit	*unit type*

Addition to the typing rules [Pie02]:

$$\Gamma \vdash \mathtt{unit}\colon \mathtt{Unit} \qquad \text{(Unit)}$$

Added derived form [Pie02]:

$$t_1; t_2 \overset{\text{def}}{=} (\lambda x\colon \mathtt{Unit}.t_2)t_1 \text{ where } x \notin FV(t_2)$$

"The function is applied to the term t_1 where the input variable x is not part of the "free variables"[2] (FV) of the term t_2".

Addition to the evaluation rules [Pie02]:

$$\frac{t_1 \longrightarrow t_1'}{t_1; t_2 \longrightarrow t_1'; t_2} \qquad \text{(Sequence)}$$

"If there is a sequence (noted by ;) and there is a step from t_1 to t_1', evaluate the term t_1".

$$\mathtt{unit}; t_2 \longrightarrow t_2 \qquad \text{(Sequence Next)}$$

"If the lefthand side of a sequence is reduced to `unit`, return the result of t_2".

Addition to the typing rules [Pie02]:

$$\frac{\Gamma \vdash t_1\colon \mathtt{Unit} \quad \Gamma \vdash t_2\colon \mathtt{T}_2}{\Gamma \vdash t_1; t_2\colon \mathtt{T}_2} \qquad \text{(Sequence)}$$

"If t_1 is of type `Unit` and t_2 has a type $\mathtt{T}_2$ then the resulting type of the sequence will be $\mathtt{T}_2$".

[2]Free variables are not bound variables in the term.

4.2 Ascription

A very handy tool for our simple programming language is the usage of ascription. It is often used for "documentation" purposes [Pie02]. It defines a way to substitute long type names with shorter ones. An example for such a substitute in the Haskell language would be: "`type MyType = Double -> Double -> [Char]`" which defines the type `MyType` as a function that takes a double and a double and returns an array of characters.

Addition to the syntax [Pie02]:

$$\begin{array}{lll} \mathtt{t} & ::= & & \textit{terms}: \\ & | & \mathtt{...} & \\ & | & \mathtt{t\ as\ T} & \textit{ascription} \end{array}$$

Addition to the evaluation rules [Pie02]:

$$v_1 \ \mathsf{as}\ \mathsf{T} \longrightarrow v_1 \qquad \text{(Ascribe Value)}$$

"The term v_1 as T returns v_1".

$$\frac{t_1 \longrightarrow t_1'}{t_1 \ \mathsf{as}\ \mathsf{T} \longrightarrow t_1' \ \mathsf{as}\ \mathsf{T}} \qquad \text{(Ascription Evaluation)}$$

"If there is a step from t_1 to t_1', evaluate the step in the syntax".

Addition to the typing rules [Pie02]:

$$\frac{\Gamma \vdash t_1 : \mathsf{T}}{\Gamma \vdash t_1 \ \mathsf{as}\ \mathsf{T} : \mathsf{T}} \qquad \text{(Ascribe)}$$

"If t_1 is assumed with the type T in the context, the term t_1 as T will yield a type T".

4.3 Let Bindings

Let bindings are a useful tool to avoid repetition in complex expressions. They are found in Haskell as well:

```
add x y =
    let result = x + y
    in
        result
```

Addition to the syntax [Pie02]:

$$\begin{array}{lll} \mathtt{t} & ::= & & \textit{terms}: \\ & | & \mathtt{...} & \\ & | & \mathtt{let\ x=t\ in\ t} & \textit{let binding} \end{array}$$

Addition to the evaluation rules [Pie02]:

$$\mathtt{let}\ x = v_1\ \mathtt{in}\ t_2 \longrightarrow [x \mapsto v_1]t_2 \qquad \text{(Let-Bind Value)}$$

"In the given abstraction t_2 replace all occurencies of x with v_1".

$$\frac{t_1 \longrightarrow t_1'}{\mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \longrightarrow \mathtt{let}\ x = t_1'\ \mathtt{in}\ t_2} \qquad \text{(Let)}$$

"If there is a step from t_1 to t_1', evaluate the step in the syntax before evaluating the let binding itself".

Addition to the typing rules [Pie02]:

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_1 \quad \Gamma, x : \mathsf{T}_1 \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : \mathsf{T}_2} \qquad \text{(Let)}$$

"To calculate the type of the let binding, calculate the type of the bound term. The bound term will yield the same type as the used term for the binding".

4.4 Pairs and Tuples

Until now, the additions were minor and added some syntactic sugar to the language of the simple typed λ-Calculus. The following simple extensions will enrich the language with features that are often found in programming languages.

Pairs - and their more general counterpart Tuples - are a construct to group values and terms together. Pairs are product types of exactly two values and therefore have slightly different evaluation rules. Tuples are the general way of pairs and therefore only the rules of tuples will be explained since they include the rules of pairs as well.

Addition to the syntax [Pie02]:

$$\begin{array}{llll} \mathtt{t} & ::= & & \textit{terms}: \\ & | & \mathtt{...} & \\ & | & \{t_i^{\,i\in 1..n}\} & \textit{tuple} \\ & | & \mathtt{t.i} & \textit{projection} \end{array}$$

$\{t_i^{\,i\in 1..n}\}$ means that for example with $i = 3$ we have a tuple of three elements. A tuple with $i = 2$ would be a pair. $\{t_i^{\,i\in 1..n}\}$ with $i = 3 \mapsto \{t_1, t_2, t_3\}$. The projection is needed to access the elements in a tuple at the given index.

$$\begin{array}{llll} \mathtt{v} & ::= & & \textit{values}: \\ & | & \mathtt{...} & \\ & | & \{v_i^{\,i\in 1..n}\} & \textit{tuple value} \end{array}$$

$$\begin{array}{llll} \mathtt{T} & ::= & & \textit{types}: \\ & | & \mathtt{...} & \\ & | & \{T_i^{\,i\in 1..n}\} & \textit{tuple type} \end{array}$$

Addition to the evaluation rules [Pie02]:

$$\{v_i^{\,i\in 1..n}\}.j \longrightarrow v_j \qquad \text{(Tuple projection)}$$

"When the projection with index j is applied to a tuple with i values, return the value at index j".

$$\frac{t_1 \longrightarrow t_1'}{t_1.i \longrightarrow t_1'.i} \qquad \text{(Projection)}$$

"If there is a step from t_1 to t_1', evaluate the step in the syntax before executing the projection".

$$\frac{t_j \longrightarrow t_j'}{\{v_i^{\,i\in 1..j-1}, t_j, t_k^{\,k\in j+1..n}\} \longrightarrow \{v_i^{\,i\in 1..j-1}, t_j', t_k^{\,k\in j+1..n}\}} \qquad \text{(Tuple)}$$

"If there is a step from t_j to t'_j, evaluate the leftmost term t_j to t'_j that is not a value". This forces the tuple to be fully evaluated before any projections can be executed on the tuple. Also, it enforces the evaluation direction for the tuple from left to write. In other terms: $\{t_1, t_2\} \mapsto \{v_1, t_2\} \mapsto \{v_1, v_2\}$.

Addition to the typing rules [Pie02]:

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : \mathsf{T}_i}{\Gamma \vdash \{t_i^{\,i\in 1..n}\} : \{\mathsf{T}_i^{\,i\in 1..n}\}} \qquad \text{(Tuple)}$$

"For each element in the tuple with index i, we calculate the type and add the whole tuple to the typing context Γ in the form $\{T_1, T_2, \ldots\}$".

$$\frac{\Gamma \vdash t_1 : \{\mathsf{T}_i^{\,i\in 1..n}\}}{\Gamma \vdash t_i.j : \mathsf{T}_j} \qquad \text{(Projection)}$$

"If the term t_1 is a tuple type with i entries, the projection $t_1.j$ will yield an element of the type T_j".

4.5 Records

Since tuples have indices and must be accessed that way, we may want to name the elements in a tuple. "Records" provide a way to label entries of a tuple and create a possibility to semantically group terms together. One could loosely compare them with `Structs` from programming languages like `GoLang`.

Addition to the syntax [Pie02]:

$$\begin{array}{lcll} \mathtt{t} & ::= & & \textit{terms}: \\ & | & \ldots & \\ & | & \{l_i = t_i^{\,i\in 1..n}\} & \textit{record} \\ & | & \mathtt{t.l} & \textit{projection} \end{array}$$

This syntax rule follows the same principle as for the tuple. One change to note is, that the projection is not done via an index but with a label l.

$$\begin{array}{lcll} \mathtt{v} & ::= & & \textit{values}: \\ & | & \ldots & \\ & | & \{l_i = v_i^{\,i\in 1..n}\} & \textit{record value} \end{array}$$

$$\begin{array}{lcll} \mathsf{T} & ::= & & \textit{types}: \\ & | & \ldots & \\ & | & \{l_i : T_i^{\,i\in 1..n}\} & \textit{tuple of records} \end{array}$$

Addition to the evaluation rules [Pie02]:

$$\{l_i = v_i^{\,i\in 1..n}\}.l_j \longrightarrow v_j \qquad \text{(Record projection)}$$

"When the projection with label j is applied to a record with i values, return the value with the label j".

$$\frac{t_1 \longrightarrow t'_1}{t_1.l \longrightarrow t'_1.l} \qquad \text{(Projection)}$$

"If there is a step from t_1 to t'_1, evaluate the step in the syntax before executing the projection".

$$\frac{t_j \longrightarrow t'_j}{\{l_i = v_i^{\,i\in 1..j-1}, l_j = t_j, l_k = t_k^{\,k\in j+1..n}\} \longrightarrow \{l_i = v_i^{\,i\in 1..j-1}, l_j = t'_j, l_k = t_k^{\,k\in j+1..n}} \qquad \text{(Record)}$$

"If there is a step from t_j to t'_j, evaluate the leftmost term $l_j = t_j$ to $l_j = t'_j$ that is not a value". This inforces the same evaluation rules on records as the above rules did on tuples. In other terms: $\{\text{foo} = t_1, \text{bar} = t_2\} \mapsto \{\text{foo} = v_1, \text{bar} = t_2\} \mapsto \{\text{foo} = v_1, \text{bar} = v_2\}$.

Addition to the typing rules [Pie02]:

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : \mathsf{T}_i}{\Gamma \vdash \{l_i = t_i^{\,i\in 1..n}\} : \{l_i : \mathsf{T}_i^{\,i\in 1..n}\}} \qquad \text{(Record)}$$

"For each element in the record with label l, we calculate the type and add the whole record to the typing context Γ in the form $\{l_1 : T_1, l_2 : T_2, \ldots\}$".

$$\frac{\Gamma \vdash t_1 : \{l_i : \mathsf{T}_i^{\,i\in 1..n}\}}{\Gamma \vdash t_i.l_j : \mathsf{T}_j} \qquad \text{(Projection)}$$

"If the term t_1 is a record type with i entries, the projection $t_1.l_j$ will yield an element of the type T_j at the position of label l_j".

4.6 Sums and Variants

Many programs need to tackle variant types. This means that we can sum together multiple shapes of a type into a summary type. Such variants are algebraic data types and are often used in functional languages with pattern matching. This paper will use the generalized definition of the variant to describe the principle. So instead of a sum type `T + T` we use the labeled variant type $\langle l_1 : \mathsf{T}_1, l_2 : \mathsf{T}_2\rangle$.

With the variant types in place, our language could now type-check and interpret the following lines of code (given in the Haskell syntax for readability):

```haskell
type Weekend = Saturday | Sunday
getName day = case day of
    Saturday -> "Saturday"
    Sunday -> "Sunday"
```

4.7 General Recursion

4.8 Lists

4.9 References

4.10 Exceptions

4.11 Summary

With those possibilities, a variety of concepts can be created. While there exists the concept of "exceptions" for example, it does not support subtyping of any kind. Polymorphism is part of a higher version of a type system, for example System F.

TODO: reference to marcs paper about systemF

The given base of the “pure simple typed λ-Calculus” and the simple extensions above now give us a language that can successfully compile and compute the following lines of (TypeScript-ish) code.

TODO: Code

Since the λ-Calculus is the mathematical foundation of several functional programming languages, the features (without polymorphism) can be translated into them. Let us review the stated features in a pure functional language like Haskell:

TODO: Code

5 Conclusion

Give the conclusion over the paper. What was shown, why it was shown.

What should the reader know about the calculus now and how the reader can translate that into typed languages

References

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6).* Princeton University Press, 1941.

[Ham18] Gary Hammock. Latex listings - javascript & es6. https://github.com/ghammock/LaTeX_Listings_JavaScript_ES6, 2018. Accessed: 2020-10-10.

[Med12] Gonzalo Medina. How do i put text over symbols? https://tex.stackexchange.com/questions/74125/how-do-i-put-text-over-symbols, 2012. Accessed: 2020-10-11.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, MA, USA, 2002.

[Sun19] Wu Sun. Use minted syntax highlighting in latex with visual studio code latex workshop. https://wusun.name/blog/2019-01-17-minted-vscode/, 2019. Accessed: 2020-10-11.

[Tur37] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.