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: Fall 2020

Abstract

Type systems play a fundamental role in modern programming languages. They provide assistance and error handling to the developer before the code hits a productive system. They help reduce the errors that can occur when two elements of different types interact with each other. This paper focuses on how one can create and add such a type system to an untyped language and create a simple typed language with some useful extensions.

Keywords: Lambda-Calculus, type systems, simple types

1 Introduction

> The road from untyped to typed universes has been followed many times, in many different fields, and largely for the same reasons.
>
> Luca Cardelli and Peter Wegner (1985)

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 of 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 - ish, TypeScript[1] - ish and Haskell[2] syntax 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 the 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 a type of compiler (i.e., transpiler) to the language.

The result of the paper will be a simply typed programming language with simple extensions. The language will contain a simple type system that can statically analyse its language. Along the sections, the different rules and explanations for them are provided. The paper does not include any variants of subtyping like polymorphism. The remainder of this paper will give further introduction into the topic, a brief overview over the λ-Calculus, an overview over the topic of "types" in general, and the application of simple types to the untyped λ-Calculus to create the simply-typed λ-Calculus as well as a list of simple extensions to the simply-typed λ-Calculus which make the language more practical and useful.

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

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

As software developers, we understand that this statement is not going to create the desired output - assuming we have an untyped language. Strings and numbers are not of the same type and cannot be subtracted from one another. 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 chapter of "Types and Programming Languages" by Benjamin C. Pierce [Pie02].

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 computation that only contains three rules of operation [Pie02]. Those three operations can be viewed in the following "grammar":

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

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

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*

The untyped λ-Calculus is Turing complete, which means it can compute *any* program. In such an untyped system two special, but unwanted, states can be achieved:

- *stuck*: the program is stuck when no more rules can be applied and therefore the program cannot run to its end
- *infinity*: the program encounters a term that reduces to a term with the exact same terms (e.g. general recursion without termination) so that the program will never reach a terminating state

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 no more valid rules can be applied.

Since it is not desirable for computer programs to run to infinity or be stuck at a point in time, 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 inputs and outputs will be counted towards "useful" programs.

3 Types and Simple Types

> Well-typed programs cannot "go wrong."
>
> Robin Milner (1978)

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 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 resulting type of f without *running* the function. Shown in typescript syntax this could mean:

```
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. However, how do we get there? We add the typing relation to the grammar and define that terms and variables must have a type:

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.

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

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

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

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

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

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 Type-Safety. 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 will not 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, the literature often substitutes them with a letter for all *unknown* base types [Pie02]. Often, constructs like $\mathcal{A}$ or other letters are seen. In this paper, we will establish the following syntactic form from Benjamin C. Pierce [Pie02]:

T	::=		*types*:
	|	. . .	
	|	A	*base type*

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:

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

A *type environment* (Γ) is introduced in the following derivation rules. This environment (or sometimes called *type context*) is a mathematical set with variables and their types [Pie02]. When our type-checker starts, the starting 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 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 $T_1 \to T_2$".

Furthermore, 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.

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

If t_1 is a function that takes T_{11} and returns 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()

// number that becomes a 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 the basic operations a computation needs. We are 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].

4.0.1 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*

4.0.2 Evaluation.

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \quad \text{(Application 1)}$$

If there exists an evaluation step from t_1 to t_1', take this step prior to the evaluation step of t_2. Since there are no other applicable rules, this forces the program to first evaluate all terms of t_1 until other rules become applicable.

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \quad \text{(Application 2)}$$

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 left-hand side of the application is reduced to a value, evaluate the right-hand side.

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

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.

4.0.3 Typing.

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

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

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

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].

4.1.1 Addition to the syntax [Pie02].

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

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

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

4.1.2 Addition to the typing rules [Pie02].

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \qquad \text{(Unit)}$$

4.1.3 Added derived form [Pie02].

$$t_1; t_2 \stackrel{\text{def}}{=} (\lambda x : \texttt{Unit}.t_2)\ t_1 \text{ where } x \notin FV(t_2)$$

The function is applied to the term t_1 where the input variable x is not part of the "free variables"[3] (FV) of the term t_2.

4.1.4 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.

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

If the left-hand side of a sequence is reduced to a `unit` value, return the result of t_2.

4.1.5 Addition to the typing rules [Pie02].

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

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

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.

4.2.1 Addition to the syntax [Pie02].

```
t   ::=          terms:
    |   ...
    |   t as T    ascription
```

4.2.2 Addition to the evaluation rules [Pie02].

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

The term v_1 as T returns v_1.

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

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

4.2.3 Addition to the typing rules [Pie02].

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

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
```

4.3.1 Addition to the syntax [Pie02].

```
t   ::=                  terms:
    |   ...
    |   let x = t in t   let binding
```

4.3.2 Addition to the evaluation rules [Pie02].

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

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

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

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

4.3.3 Addition to the typing rules [Pie02].

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

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.

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

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.

4.4.1 Addition to the syntax [Pie02].

t	::=		*terms*:
	|	...	
	|	$\{t_i^{i\in 1..n}\}$	*tuple*
	|	t.i	*projection*

$\{t_i^{i\in 1..n}\}$ 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.

v	::=		*values*:
	|	...	
	|	$\{v_i^{i\in 1..n}\}$	*tuple value*

T	::=		*types*:
	|	...	
	|	$\{T_i^{i\in 1..n}\}$	*tuple type*

4.4.2 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, then return the value with 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. Moreover, it enforces the evaluation direction for the tuple from left to write. In other terms: $\{t_1, t_2\} \mapsto \{v_1, t_2\} \mapsto \{v_1, v_2\}$.

4.4.3 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 $\{T_1, T_2, \dots\}$.

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

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

4.5.1 Addition to the syntax [Pie02].

t	::=		*terms*:
	|	...	
	|	$\{l_i = t_i^{i\in 1..n}\}$	*record*
	|	t.l	*projection*

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.

v	::=		*values*:
	|	...	
	|	$\{l_i = v_i^{i\in 1..n}\}$	*record value*

T	::=		*types*:
	|	...	
	|	$\{l_i : T_i^{i\in 1..n}\}$	*tuple of records*

4.5.2 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.

$$\frac{t_j \longrightarrow t_j'}{\begin{array}{l}\{l_i = v_i^{i\in 1..j-1}, l_j = t_j, l_k = t_k^{k\in j+1..n}\} \\ \longrightarrow \{l_i = v_i^{i\in 1..j-1}, l_j = t_j', l_k = t_k^{k\in j+1..n}\}\end{array}} \qquad \text{(Record)}$$

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 enforces the same evaluation rules on records as the above rules did on tuples. In other terms: $\{\text{foo} = t_1, \text{bar} = t_2\} \mapsto \{\text{foo} = v_1, \text{bar} = t_2\} \mapsto \{\text{foo} = v_1, \text{bar} = v_2\}$.

4.5.3 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 $\{l_1 : T_1, l_2 : T_2, \dots\}$.

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

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

4.6 Sums and Variants

Many programs need to tackle variants. 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 for pattern matching. One can compare them vaguely to *Enums* of object-oriented languages like C#. This paper will use the generalized definition of the variant to describe the principle. Thus, instead of a sum type `T + T`, we use the labeled variant type $\langle l_1 : \mathsf{T}_1, l_2 : \mathsf{T}_2 \rangle$. The sum type could be compared to Haskell's "*Either a b*" type, which can be either type "a" or "b".

4.6.1 Addition to the syntax [Pie02].

`t` ::=		*terms*:
|	`. . .`	
|	`<l=t> as T`	*tagging*
|	`case t of` $<l_i = x_i> \Rightarrow t_i^{i \in 1..n}$	*case*

This syntax allows generalized labeled variants of types.

`T` ::=		*types*:
|	`. . .`	
|	$<l_i : T_i^{i \in 1..n}>$	*type of variants*

4.6.2 Addition to the evaluation rules [Pie02].

$$\texttt{case}(<l_j = v_j> \texttt{ as T}) \texttt{ of } <l_i = x_i> \Rightarrow t_i^{i \in 1..n} \longrightarrow [x_j \mapsto v_j]t_j \qquad \text{(Case variant)}$$

Check the variant with a `case variant` syntax and return the given term to the right-hand side of the arrow. Replace the x variable in the term with the ascribed type.

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case } t_0 \texttt{ of } <l_i = x_i> \Rightarrow t_i^{i \in 1..n} \longrightarrow \texttt{case } t_0' \texttt{ of } <l_i = x_i> \Rightarrow t_i^{i \in 1..n}} \qquad \text{(Case)}$$

If there is a step from t_0 to t_0', evaluate the term in the case clause before applying any mapping.

$$\frac{t_i \longrightarrow t_i'}{<l_i = t_i> \texttt{ as T} \longrightarrow <l_i = t_i'> \texttt{ as T}} \qquad \text{(Variant)}$$

If there is a step from t_i to t_i', evaluate the term in the variant.

4.6.3 Addition to the typing rules [Pie02].

$$\frac{\Gamma \vdash t_j : \mathsf{T}_j}{\Gamma \vdash <l_j = t_j> \texttt{ as } <l_i : T_i^{i \in 1..n}> : <l_i : T_i^{i \in 1..n}>} \qquad \text{(Variant)}$$

When the type of t_j is in the typing environment, add the labeled variant types in the environment as well with their corresponding labels and term types.

$$\frac{\Gamma \vdash t_0 : <l_i : \mathsf{T}_i^{i \in 1..n}> \quad \text{for each } i\ \Gamma, x_i : \mathsf{T}_i \vdash t_i : T}{\Gamma \vdash \texttt{case } t_0 \texttt{ of } <l_i = x_i> \Rightarrow t_i^{i \in 1..n} : \mathsf{T}} \qquad \text{(Case)}$$

If t_0 is a variant with i label (and therefore variants), the 'case of' syntax will return the specific type of the variant instead of the summary type.

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):

```
data StringOrInt = MyString String | MyInt Int
getStringValue :: StringOrInt -> String
getStringValue value = case value of
    MyString s -> s
    MyInt i -> show i
```

Variants are often used to represent variable return values. One can think of the "Option" type in F# or the "Maybe" type of Haskell. Both define two variants of a result, namely "Some" ("Just") or "None" ("Nothing"). A computation that may have a none result can return this type constructor of the variant and can signal an empty or faulty result. A typical case could be number parsing. When one wants to parse the string `"12"` into a number, the result in Haskell could be `"Just 12"`, but on the other hand, parsing `"12i"` would result in `"Nothing"`.

4.7 General Recursion

This section focuses on the general recursion. In the untyped λ-Calculus, a general recursion can be solved by a fixed-point combinator which "unrolls" the recursion of a function [Pie02].

This fixed-point combinator is further called "`fix`".

For the sake of completeness, the formal definition of the fixed combinator is stated below [Pie02]: TODO: Ref to simons paper (hopefully he has the general recursion covered)

$$\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)) \qquad \text{(Y combinator)}$$

The given function is called the "paradoxical Y combinator by Haskell B. Curry". It is an implementation of such a `fix` combinator. The combinator returns a fixed point of a

function if any exists by applying the function to itself. To give a better understanding of how this recursion works, the following lines should show the progress of the reduction of the fix combinator:

$$\begin{aligned} Y\,g &= (\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,g \\ &= (\lambda x.\,g(x\,x))\,(\lambda x.\,g(x\,x)) \\ &= g((\lambda x.\,g(x\,x))\,(\lambda x.\,g(x\,x))) \\ &= g(Y\,g) \end{aligned}$$

The Y combinator is "solved" by β-reduction and equality rules. When the equality is applied multiple times the following equation emerges:

$$Y\,g = g(Y\,g) = g(g(Y\,g)) = \ldots$$

Given the terms of the language we constructed so far, the fix combinator could be defined as: TODO: get citation for this

$$\texttt{fix}\ f = \texttt{let}\ x = f\,x\ \texttt{in}\ x$$

To understand the impact of the `fix` combinator in the typed universe, let us analyze the factorial equation:

$$f(x) = \begin{cases} 1 & \text{for } x \in \{0,1\} \\ x * f(x-1) & \text{for } x \in \mathbb{N} \setminus \{0,1\} \end{cases}$$

(Recursive Factorial)

This definition translated into an untyped λ-Calculus syntax would be [Pie02]:

```
factorial = λn.
            if n = 0 then 1
            else n * factorial(pred(n))
```

With this definition at hand, the fix operator can now unroll the recursion and create a function that does those "if" comparisons until the termination point is reached. In general, the "fix" operator takes a recursive function (generator) and creates a fixed point function that unrolls the function call to itself until the end is reached. The resulting function states as follows [Pie02]:

```
if n=0 then 1
else n * (if (n-1)=0 then 1
          else (n-1) * (if (n-2)=0 then 1
                        else ...))
```

The given `fix` function has a big problem in our narrowed down universe of "simple types". Since the function is able to create an endless recursion, it is not valid in our context. All functions must eventually terminate to adhere to the given rules of a typed system. The only applicable solution for now[4] is to define `fix` as a primitive in the language and use typing rules to mimic the behavior [Pie02].

[4] As long as we only have "simple types".

4.7.1 Addition to the syntax [Pie02].

t	::=	*terms*:
	| `...`	
	| `fix t`	*fixed point of t*

4.7.2 Addition to the evaluation rules [Pie02].

$$\texttt{fix}\ (\lambda x{:}\,\mathsf{T}_1.t_2) \longrightarrow [x \mapsto (\texttt{fix}\ (\lambda x{:}\,\mathsf{T}_1.t_2))]t_2$$

(Fix Beta Reduction)

When applying the `fix` function to a given term t_2, replace all occurrences of the bound variable (x) with the term itself.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'} \qquad \text{(Fix)}$$

If there is a step from t_1 to t_1', evaluate the term before applying the `fix` function to it.

4.7.3 Addition to the typing rules [Pie02].

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_1 \longrightarrow \mathsf{T}_1}{\Gamma \vdash \texttt{fix}\ t_1 : \mathsf{T}_1} \qquad \text{(Variant)}$$

If the type of t_1 is in the context and has a function type $\mathsf{T} \longrightarrow \mathsf{T}$, then the application of `fix` to t_1 will yield the type T_1.

4.7.4 Added derived form [Pie02].

$$\begin{aligned} &\texttt{letrec}\ x{:}\,\mathsf{T}_1 = t_1\ \texttt{in}\ t_2 \\ \overset{\text{def}}{=}\ &\texttt{let}\ x = \texttt{fix}\ (\lambda x{:}\,\mathsf{T}_1.t_1)\ \texttt{in}\ t_2 \end{aligned}$$

Define the form `letrec...` as a "let binding" with the application of the `fix` function to the term in t_2.

4.8 Lists

We have seen some "base types" like `Nat` or `Bool` and "type constructors" records and variant types which build new types out of old ones [Pie02]. To complete the list - pun intended - we introduce "lists" here. A list is a practical and useful type constructor that describes a finite set of elements which are fetched from the type of the list. In addition to the list definition itself, some useful helper methods come along with it to make the list usable in a "practical" way.

4.8.1 Addition to the syntax [Pie02].

t	::=	*terms*:
	| `...`	
	| `nil[T]`	*empty list*
	| `cons[T] t t`	*list constructor*
	| `isnil[T] t`	*test for empty list*
	| `head[T] t`	*head of a list*
	| `tail[T] t`	*tail of a list*

```
v  ::=                values:
   |  ...
   |  nil[T]         empty list
   |  cons[T] v v     list constructor

T  ::=                types:
   |  ...
   |  List T         type of lists
```

4.8.2 Addition to the evaluation rules [Pie02].

$$\frac{t_1 \longrightarrow t_1'}{\texttt{cons[T]}\ t_1\ t_2 \longrightarrow \texttt{cons[T]}\ t_1'\ t_2} \qquad \text{(Cons Left)}$$

If there is a step from t_1 to t_1', evaluate the term t_1 before the other terms or the constructor.

$$\frac{t_2 \longrightarrow t_2'}{\texttt{cons[T]}\ v_1\ t_2 \longrightarrow \texttt{cons[T]}\ v_1\ t_2'} \qquad \text{(Cons Right)}$$

If there is a step from t_2 to t_2' and the left-hand side is already reduced to a value, evaluate the term t_2 before the list constructor.

$$\texttt{isnil[S]}\ (\texttt{nil[T]}) \longrightarrow \texttt{true} \qquad \text{(IsNil of Nil)}$$

The application of `isnil` to an empty list constructed with `nil[]` must return `true`. TODO: why S in isnil? and not T?

$$\texttt{isnil[S]}\ (\texttt{cons[T]}v_1v_2) \longrightarrow \texttt{false} \qquad \text{(IsNil of Nil)}$$

The application of `isnil` to a nonempty list constructed with `cons[]` and two values must return `false`.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{isnil[T]}\ t_1 \longrightarrow \texttt{isnil[T]}\ t_1'} \qquad \text{(IsNil)}$$

If there is a step from t_1 to t_1', evaluate the term t first until it is a value before evaluating the `isnil` function.

$$\texttt{head[S]}\ (\texttt{cons[T]}\ v_1v_2) \longrightarrow v_1 \qquad \text{(Head of cons)}$$

If the function `head` is applied to a list of values, it returns the left-hand element (head) of the list.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{head[T]}\ t_1 \longrightarrow \texttt{head[T]}\ t_1'} \qquad \text{(Head)}$$

If there is a step from t_1 to t_1', evaluate t prior to applying the head function to the term.

$$\texttt{tail[S]}\ (\texttt{cons[T]}\ v_1v_2) \longrightarrow v_2 \qquad \text{(Tail of cons)}$$

If the function `tail` is applied to a list of values, it returns the right-hand element (tail) of the list.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{tail[T]}\ t_1 \longrightarrow \texttt{tail[T]}\ t_1'} \qquad \text{(Tail)}$$

If there is a step from t_1 to t_1', evaluate t prior to applying the tail function to the term.

4.8.3 Addition to the typing rules [Pie02].

$$\Gamma \vdash \texttt{nil[T}_1\texttt{]} : \texttt{List T}_1 \qquad \text{(Nil)}$$

The calculated type of the list `nil[T]` is `List T`.

$$\frac{\Gamma \vdash t_1 : \texttt{T}_1 \quad \Gamma \vdash t_2 : \texttt{List T}_1}{\Gamma \vdash \texttt{cons[T}_1\texttt{]}\ t_1\ t_2 : \texttt{List T}_1} \qquad \text{(Constructor)}$$

If t_1 is of type $\texttt{T}_1$ and t_2 is a list of $\texttt{T}_1$, then the cons(tructor) of a list with those two terms will also create a list of type $\texttt{T}_1$. This essentially allows the list constructor to create consecutive lists of terms. Lists are not only limited to two elements. This can be compared to the Haskell notation of the ':' (cons) operator, which allows the creation of lists:

```
-- this creates the list [1,2,3]
list = 1:2:3:[]
```

$$\frac{\Gamma \vdash t_1 : \texttt{List T}_{11}}{\Gamma \vdash \texttt{isnil[T}_{11}\texttt{]}\ t_1 : \texttt{Bool}} \qquad \text{(IsNil)}$$

If the term t_1 is a `List T`, then the application of `isnil[T]` to this term t_1 must yield a `Bool` type.

$$\frac{\Gamma \vdash t_1 : \texttt{List T}_{11}}{\Gamma \vdash \texttt{head[T}_{11}\texttt{]}\ t_1 : \texttt{T}_{11}} \qquad \text{(Head)}$$

If the term t_1 is a `List T`, then the application of `head[T]` to this term t_1 must yield a type `T`.

$$\frac{\Gamma \vdash t_1 : \texttt{List T}_{11}}{\Gamma \vdash \texttt{tail[T}_{11}\texttt{]}\ t_1 : \texttt{T}_{11}} \qquad \text{(Tail)}$$

If the term t_1 is a `List T`, then the application of `tail[T]` to this term t_1 must yield a type `T`.

4.9 References

Up until now, all previous extensions to the "simply typed λ-Calculus" were of *pure* nature [Pie02]. When we consider imperative and object-oriented programming languages, most of them contain some mechanism to assign values to a variable. This can (and is) used to perform side effects in the execution of a function. Those side effects are often referred to as "computational effects" [Pie02]. Let the following lines of code inspire the idea behind the side effects.

```
let counter = 0;

function update(): number {
    return counter++;
}

console.log(counter());
console.log(counter());
console.log(counter());
```

One of the substantial differences between pure functional programming languages and object-oriented ones is the "referential transparency" which essentially states that given an input x, the function $f(x)$ must yield the same result each time it is called with x. In essence, one could replace the function with its result and does not change the result of the program. References are a way to introduce such desired side effects to programming languages like seen in the code lines above.

Note that most of the time the "=" sign is used for assigning values. It should be stated that instead of "equals" it should be regarded as "becomes". Because it is not a mathematical equality but more an assignment to a label.

The program needs to keep track of those references and the stored values. For this, we introduce stores with locations. As one can imagine, the store contains the real value which is encoded in the binary format of the type. The set of store locations is named $\mathcal{L}$ and the store itself is a partial function from locations l to their values [Pie02]. Another metavariable μ is used to range over stores. A reference is essentially a location in a store. To type the stores correctly, another type context is added to the language with the symbol Σ. It contains the types of locations in the store.

Some of the basic application and abstraction rules that were stated in the previous sections need to be adjusted to take stores into account.

The basic operations of references are [Pie02]:

- Assignment
- Allocation
- Dereferencing

4.9.1 Addition to the existing evaluation rules [Pie02].

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \quad \text{(Application 1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \quad \text{(Application 2)}$$

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

The given changes define that the application step contains a starting and ending point for the stores μ. The abstraction, however, does not change the store and therefore μ does not transition to μ'. The additions extend the applications so that they *must* take a store and return a new store at the end of the evaluation.

4.9.2 Addition to the existing typing rules [Pie02].

$$\Gamma \mid \Sigma \vdash \texttt{unit} : \texttt{Unit} \quad \text{(Nil)}$$

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

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

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

Now all existing rules are aware of typings that are not in the typing context Γ but also look in type environment of "Σ" for a given type.

4.9.3 Addition to the syntax [Pie02].

t	::=		*terms*:
	|	`...`	
	|	`ref t`	*reference creation*
	|	`!t`	*dereference*
	|	`t := t`	*assignment*
	|	`l`	*store location*

T	::=		*values*:
	|	`...`	
	|	`l`	*store location*

T	::=		*types*:
	|	`...`	
	|	`Ref T`	*type of references*

μ	::=		*stores*:
	|	$\varnothing$	*empty store*
	|	μ, `l = v`	*location binding*

Σ	::=		*store typings*:
	|	$\varnothing$	*empty store typing*
	|	Σ, `l : T`	*location typing*

4.9.4 Addition to the evaluation rules [Pie02].

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad \text{(Ref Value)}$$

If l is not in the domain of μ, the `ref` syntax of a value creates a new location in the store with a function to retrieve the value from the store with the given location.

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{ref}\ t_1 \mid \mu \longrightarrow \texttt{ref}\ t_1' \mid \mu'} \quad \text{(Reference)}$$

If there is a step from t_1 to t_1' and therefore a step from store μ to μ', evaluate those terms before performing the `ref` function.

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \qquad \text{(Dereference location)}$$

If the location l is in the store μ and it yields the value v, the application of $!l$ to the store will yield the value v and the same store since it is not manipulated.

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \qquad \text{(Dereference)}$$

If there is a step from t_1 to t_1' and therefore a step from the store μ to μ', evaluate the term before performing the de-referentation of the location.

$$l \ \texttt{:=}\ v_2 \mid \mu \longrightarrow \texttt{unit} \mid [l \mapsto v_2]\,\mu \qquad \text{(Assignment)}$$

The assignment of l 'becomes' v_2 will return a `unit` value and replaces l in the store μ with the value v_2.

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 \ \texttt{:=}\ t_2 \mid \mu \longrightarrow t_1' \ \texttt{:=}\ t_2 \mid \mu'} \qquad \text{(Assignment 1)}$$

If there is a step from t_1 to t_1' and therefore a step from the store μ to μ', evaluate the term t_1 before performing the assignment.

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 \ \texttt{:=}\ t_2 \mid \mu \longrightarrow v_1 \ \texttt{:=}\ t_2' \mid \mu'} \qquad \text{(Assignment 2)}$$

If there is a step from t_2 to t_2' and the location is already a value v_1, evaluate the term t_2 before performing the assignment.

4.9.5 Addition to the typing rules [Pie02].

$$\frac{\Sigma(l) = \mathsf{T}_1}{\Gamma \mid \Sigma \vdash l : \texttt{Ref}\ \mathsf{T}_1} \qquad \text{(Location)}$$

If the stored typing of l in the stored typing Σ has the type T_1, then l has the type `Ref` T_1.

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \mathsf{T}_1}{\Gamma \mid \Sigma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ \mathsf{T}_1} \qquad \text{(Reference)}$$

If the stored typing of t_1 is T_1, then the resulting type of `ref` t_1 is `Ref` T_1.

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Ref}\ \mathsf{T}_{11}}{\Gamma \mid \Sigma \vdash !t_1 : \mathsf{T}_{11}} \qquad \text{(De-Reference)}$$

If the stored typing of t_1 is a reference type `Ref` T_{11}, the 'deref' `!t` operation will yield a type T_{11}.

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Ref}\ \mathsf{T}_{11} \quad \Gamma \mid \Sigma \vdash t_2 : \mathsf{T}_{11}}{\Gamma \mid \Sigma \vdash t_1 \ \texttt{:=}\ t_2 : \texttt{Unit}} \qquad \text{(Assignment)}$$

If t_1 is a reference type of type T_{11} and t_2 is a term with type T_{11}, then the assignment t_1 `:=` t_2 will yield a `Unit` type result.

4.10 Exceptions

The final addition (i.e., simple extension) to the simply typed λ-Calculus in this paper are exceptions. With the possibility of references and general recursive functions, the need of error handling arises. Nearly all real-world programming languages have some way of signaling that the function in question is not able to carry out a given task [Pie02]. In practical terms, this could be a division by zero. As seen in 4.6, a function could just return a variant that is either a result or some other variant of the type, but when something is truly exceptional, the language should be able to throw an exception.

The following additions define the possibility to raise exceptions, as well as a way of handling them when they occur.

4.10.1 Addition to the syntax [Pie02].

`t`	::=	*terms*:
	| `...`	
	| `raise t`	*raise an exception*
	| `try t with t`	*handle exceptions*

4.10.2 Addition to the evaluation rules [Pie02].

$$(\texttt{raise}\ v_{11})t_2 \longrightarrow \texttt{raise}\ v_{11} \qquad \text{(Application Raise 1)}$$

If the left-hand side of an application does raise some error value, ignore the right-hand side and raise the given error.

$$v_1(\texttt{raise}\ v_{21}) \longrightarrow \texttt{raise}\ v_{21} \qquad \text{(Application Raise 2)}$$

If the left-hand side is reduced to a value and the right-hand side does raise an error, raise that error. Those two rules guarantee that if either side of an application does raise an exception, the error will take precedence over the other term and/or value.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{raise}\ t_1 \longrightarrow \texttt{raise}\ t_1'} \qquad \text{(Raise)}$$

If there is a step from t_1 to t_1', evaluate the term before executing the `raise` application.

$$\texttt{raise}\ (\texttt{raise}\ v_{11}) \longrightarrow \texttt{raise}\ v_{11} \qquad \text{(Re-Raise)}$$

If `raise` is applied to a term that raises a value itself, re-raise that value.

$$\texttt{try}\ v_1\ \texttt{with}\ t_2 \longrightarrow v_1 \qquad \text{(Try Value)}$$

If a term is evaluated to a value and no error is risen, return this value and ignore the term t_2.

$$\texttt{try raise}\ v_{11}\ \texttt{with}\ t_2 \longrightarrow t_2 v_{11} \qquad \text{(Try Error)}$$

If a term is evaluated and raises an error, use the term t_2 (the exception handler) and apply it to the value v_{11}.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{try } t_1 \texttt{ with } t_2 \longrightarrow \texttt{try } t_1' \texttt{ with } t_2} \qquad \text{(Try)}$$

If there is a step from t_1 to t_1', evaluate the term before evaluating other elements like the error handler.

4.10.3 Addition to the typing rules [Pie02].

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_{exn}}{\Gamma \vdash \texttt{raise } t_1 : \mathsf{T}} \qquad \text{(Exception)}$$

If t_1 is a type that can be raised (i.e., it is additional information to the error), then the whole expression can be given the type T since it may be required by the context.

$$\frac{\Gamma \vdash t_1 : \mathsf{T} \quad \Gamma \vdash t_2 : \mathsf{T}_{exn} \longrightarrow \mathsf{T}}{\Gamma \vdash \texttt{try } t_1 \texttt{ with } t_2 : \mathsf{T}} \qquad \text{(Try)}$$

If the first term t_1 is of type T and the second term t_2 is an exception handler of type $\mathsf{T}_{exn} \longrightarrow \mathsf{T}$, then the resulting type of the '`try ... with ...`' expression will be of type T.

5 Conclusion

With the application of simple types to the λ-Calculus and the addition of "simple extensions", the result of this paper is a form of the "simply-typed λ-Calculus" with extensions to the language and the type system to make a practical language. It shows the additions needed and the steps taken to add a type system to an untyped language. It gives some insight into the very complex topic of type systems and how they work and provides explanations for the various mathematical formulas and derivation trees. It should be noted that the result is not a fully fledged programming language since subtyping and polymorphism are still missing which is a big part of modern languages. The simply-typed λ-Calculus, however, is a good example for type-science since it is not too complex and can be extended quite easily.

Appendix

Below, the described additions and extensions will be shown in TypeScript syntax as well as Haskell syntax to give a practical example of the created type system. Not all elements are translateable 1:1.

TypeScript

```ts
// Base Types
const foo: string = 'bar';
const nr: number = 42;

// Function Types
const f = () => console.log('');
const fx = x => x * x;

// Unit Type
function f(): void {}

// Sequencing
console.log(42);
console.log(1337);

// Ascription
const foo = 'something' as string;

// Let Bindings
let foo = (x, y) => x+y; // let
const result = foo(1,2) // "in" ...

// Pairs and Tuples
const foo = [42, 'something', true];

// Records
const foo = {name: 'Max', lastname: 'Muster'};

// Sums and Variants
type MyVariant = string | number; // Union Type

// General recursion
const factorial = (n) => n <= 1
    ? 1
    : n * factorial(n-1)

// Lists
const arr: string[] = ['one', 'two', 'three'];

// References
let variable = 1;
const f = () => variable = variable + 1;
f();

// Exceptions
throw 'Something went wrong';
```

Haskell

```hs
-- Base Types
foo = "string"
nr = 42

-- Function Types
f x y = x + y

-- Unit Type
-- Zero element tuple
empty = ()

-- Sequencing
main = do
    a <- something
    b <- something
```

```
    return 0

-- Ascription
type MyType = Double * [Char]

-- Let Bindings
f x y =
    let calc a b = a + b
    in calc x y

-- Pairs and Tuples
pair = (1, 2)
tuple = (1, 2, 3)

-- Records
data Person = Person
    { name :: String
    , lastname :: String
    }

-- Sums and Variants
data Maybe a = Just a | Nothing
data Weekdays = Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
    | Sunday

-- General recursion
factorial 0 = 1
factorial 1 = 1
factorial n = n * (factorial n-1)

-- Lists
list = 1 :: 2 :: 3 :: []

-- References
-- Doable with Monads
-- for example with the State-Monad

// Exceptions
error "Something went wrong"
```

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.