Aalto University
School of Electrical Engineering
Degree Programme in Radio Science and Engineering

Oleg Grenrus

Domain specific type systems

Master's Thesis
Espoo, October 10, 2016

Supervisor: Professori Jorma Tarhio
Advisor: Gergely Patai (M.Sc.)

Aalto University
School of Electrical Engineering
Degree Programme in Radio Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Oleg Grenrus

Title:
Domain specific type systems

Date: October 10, 2016 **Pages:** vii + 78 + X

Major: Software Technology **Code:** T-106

Supervisor: Professor Jorma Tarhio

Advisor: Gergely Patai (M.Sc.)

Type system tailored for specific domain could radically improve quality of the program. Many domains have natural types, yet they are difficult to encode in mainstream languages' type systems. If the encoding is possible, it's luckily to be very troublesome to work with. We investigate a single approach of developing domain specific type systems and type checking and inference algorithms for them, and apply it to *MATLAB* and *JavaScript* programs.

Keywords: type system, type inference, domain specific language

Language: English

Aalto-yliopisto
Sähkötekniikan korkeakoulu
Radiotieteiden ja -tekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Oleg Grenrus		
Työn nimi:	Sovelluskohtaiset tyyppijärjestelmät		
Päiväys:	10. lokakuuta 2016	**Sivumäärä:**	vii + 78 + X
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Jorma Tarhio		
Ohjaaja:	Gergely Patai (M.Sc.)		

Tyyppijärjestelmä, joka on suunniteltu tiettyä sovellusaluetta varten, voi merkittävästi parantaa sovelluksen laatua. Monilla sovellusalueilla on luonnollisia tyyppejä, joita ei ole helppo ilmaista yleiskäyttöisten ohjelmointikielten tyyppijärjestelmissä. Ja vaikka se olisikin mahdollista, koodaus ei ole välttämättä luonteva. Tässä tutkielmassa tarkastellaan erästä tapaa suunnitella ja toteuttaa sovelluskohtaisia tyyppijärjestelmiä ja niiden tyyppitarkistus- ja tyyppipäättelyalgoritmeja. Sovellamme menetelmää *Matlab*- ja *JavaScript*-kielillä kirjoitettuihin ohjelmiin.

Asiasanat:	tyyppijärjestelmä, tyyppipäättely
Kieli:	Englanti

Acknowledgements

I would like to thank my advisor Gergely Patai for all the discussions we had and the comments he made to improve this thesis. I thank Prof. Jorma Tarhio for supervising my work. I also thank all my friends who regularly kept asking me about the state of the thesis. Especially, Santtu, who kept motivating me every day.

Espoo, October 10, 2016

Oleg Grenrus

Contents

List of Figures

Chapter 1

Introduction

Recently there is a lot of development to introduce general-purpose type systems to general-purpose languages: Flow, TypeScript (JavaScript), Hack (PHP), core.typed (Clojure), Typed Racket (Scheme) [9, 26–28, 67]. This thesis takes a different route, we try to create a domain specific type system for a particular domain or program.

Creating a sound type system is a non-trivial task. To overcome that, we take an expressive enough existing type-system as a basis, but use only a subset of it. As we work in a particular domain, we have enough information to elaborate types. This is contrary to usual academic practice, where new features are *added* to existing type systems.

In this work we investigate one approach of implementing domain specific type systems, and the type-inference implementations for them. As an empirical proof, we use the method twice: for MATLAB and JAVASCRIPT programs. The approach is mostly theoretical, there are only proof of concept implementations of the type checkers. Further work is needed to make them ready for production use, and to investigate possible implementation problems.

We will introduce notation and basic results about type-systems in Chapter 2. In particular we describe simply typed lambda calculus, System F and dependent type systems. Also we discuss how the type-inference algorithms can be implemented in Chapter 3.

In Chapter 4 we develop a type system to a subset of MATLAB language. Cleve Moler developed MATLAB [51], so his students don't need to learn FORTRAN to use powerful linear algebra routines. MATLAB is originally a domain specific language, though it evolved into a more general and

powerful language.

MATLAB is a dynamically typed language, yet applied linear algebra has a natural type language: we operate on matrices of different sizes. We omit types while writing mathematical expressions, as the types and matrix dimensions could be inferred easily.

Quite often we, the developers, make errors. It's frustrating to get a matrix dimensions mismatch run-time error, especially in the middle of long running computations. These mistakes can be easily caught by the type-checker, even before a program is run.

As the language of linear algebra is quite simple, we conduct a more challenging experiment in Chapter 5. We develop a type system to express types in existing JAVASCRIPT program: JSVERIFY [42]. We show that the approach of writing domain specific elaborator is possible for complicated programs as well.

The development is made using the functional programming language Haskell [39, 46], which has also influenced this work.

Chapter 2

Lambda calculus

In this chapter we introduce untyped and simply typed lambda calculi and basic results about them. [5, 6, 59]. After that we show more sophisticated type systems, such as System F and dependent type systems.

2.1 Untyped lambda calculus

Definition 2.1. The set of λ-terms is built up from an infinite set of variables $V = \{x, y, z, \ldots\}$ using application and abstraction. Using abstract syntax we can write describe the terms as:

λ-CALCULUS TERM	e	$:= x$	*variable*
		$\mid \lambda x.\, e$	*abstraction*
		$\mid f\ x$	*application*

Definition 2.2. The set of *free variables* of a term t, written $FV(t)$, is defined as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.\, e) &= FV(e) \setminus \{x\} \\ FV(f\ x) &= FV(f) \cup FV(x) \end{aligned}$$

Definition 2.3. The term is said to be *closed*, if the set of free variables is empty.

$$\begin{aligned} FV(\lambda x.\, x) = \emptyset &\Rightarrow \lambda x.\, x \text{ is closed} \\ FV(\lambda x.\, f\ x) = \{f\} &\Rightarrow \lambda x.\, f\ x \text{ is not closed} \end{aligned}$$

Definition 2.4. The result of *substitution* of N for (the free occurrence of) x in M, notation $[x/N]M$, is defined as follows: Below $x \neq y$.

$$\begin{aligned}
[x/N]\, x &= N \\
[x/N]\, y &= y \\
[x/N]\, (P\ Q) &= ([x/N]\, P)\ ([x/N]\, Q) \\
[x/N]\, (\lambda y.\, P) &= \lambda y.\, ([x/N]\, P) \\
[x/N]\, (\lambda x.\, P) &= \lambda x.\, P
\end{aligned}$$

The mnemonic, "substitute x with N in M".

Definition 2.5. The principal *evaluation* axiom scheme of the λ-calculus is

$$(\lambda x.\, M)N = [x/N]\, M$$

for all λ-terms M, N. This operation is also called β-reduction, and $(\lambda x.\, M)N$ -term, β-redex, as it can be β-reduced.

It turns out that untyped λ-calculus is a powerful model of computation, being equivalent to Turing machines [65]. Basically, everything which is computable with a Turing machine is also computable using λ-calculus. The Turing machine is an intuitive model of computation, it's easy to imagine an infinite tape and a head reading and writing symbols from the tape. In that sense, lambda calculus is very abstract. It's much harder to imagine how substitution could be implemented by a mechanical device.

We need to use some kind meta language to describe Turing machines. On the other hand, the λ-calculus is quite close to modern (functional) programming languages. Therefore results about λ-calculus are easier to transfer into "mainstream" programming languages.

Take for an example HASKELL [39, 46], we can view it as an empirical proof to the above statement. If we omit types from HASKELL programs (which we can for simple programs), it will resemble the bare λ-calculus. In addition HASKELL has syntax sugar constructs, to make the language more practical.

Example 2.6 (Haskell and λ-calculus)**.** Consider a simple program for calculation of the value of a second order polynomial.

$$\begin{aligned}
sq &= \lambda x.\, mult\ x\ x \\
poly2 &= \lambda a.\, \lambda b.\, \lambda c.\, \lambda c.\, \lambda x.\, plus\ (mult\ a\ (sq\ x))\ (plus\ (mult\ b\ x)\ c)
\end{aligned}$$

The equivalent HASKELL program is very similar, only the syntax for lambda abstraction is changed from $\lambda x.\, M$ to `\x -> M`. Also it's possible to abstract over multiple variables simultaneously, i.e. $\lambda x.\, \lambda y.\, f\ x\ y$ can be written as `\x y -> f x y`.

```
sq    = \x -> mult x x
poly2 = \a b c x -> plus (mult a (sq x)) (plus (mult b x) c)
```

To mention some of the syntax sugar of HASKELL: It is possible to define infix operators, for example to make writing arithmetic expressions more convenient. Also top-level function definitions can be written in equational style: `sq = \x -> mult x x` can be written as `sq x = mult x x`. Using these syntax extensions, we can rewrite above example as:

```
sq x          = mult x x
poly2 a b c x = a * sq x + b * x + c
```

We can take this even further and prettify the typeset Haskell-code, then it will look very much like λ-calculus:

$$\begin{aligned} &\mathit{sq}\ x &&= \mathit{mult}\ x\ x \\ &\mathit{poly2}\ a\ b\ c\ x &&= a * \mathit{sq}\ x + b * x + c \end{aligned}$$

Especially with syntax extension λ-calculus is already viable as programming language. One missing part is an equivalence of expressions.

Definition 2.7 (α-equivalence)**.** Two expressions are α-equivalent if one can be obtained from the other by non-clashing substitution of free variable names.

Example 2.8. $\lambda x.\, \lambda y.\, x\ y\ z$ is α-equivalent to $\lambda a.\, \lambda b.\, a\ b\ z$, but $\lambda x.\, \lambda y.\, x\ y\ z$ is equivalent to neither $\lambda a.\, \lambda b.\, a\ b\ c$ nor $\lambda a.\, \lambda a.\, a\ a\ z$.

Example 2.9 (Church-encoding)**.** We can represent data as expressions in λ-calculus using Church encoding. The Church encoding is not intended as a practical implementation of primitive data types. Its use is to show that other primitive data types are not required to represent any calculation. [45]

Let us consider booleans: *true* and *false*. We can represent them as expressions:

$$\begin{aligned} &\mathit{true}\ t\ f &&= t \\ &\mathit{false}\ t\ f &&= f \end{aligned}$$

In ordinary programming languages we have conditionals, some kind of `if` expression or statement taking the condition, the consequence and the alternative, for example the ternary-operator in JAVASCRIPT:

```
result = condition ? consequence : alternative;
```

With Church-encoded booleans the condition itself acts as an conditional-expression:

$$result = condition\ consequence\ alternative$$

Example 2.10 (Functions on Church-encoded data)**.** It's possible to write functions working with Church-encoded data. Continuing with the booleans defined in the previous example, we can write the *and*-function:

$$and\ a\ b\ t\ f = a\ (b\ t\ f)\ f$$

This example would work in both HASKELL:

$$\begin{aligned}
&true\ t\ f = t\\
&false\ t\ f = f\\
&and'\ a\ b\ t\ f = a\ (b\ t\ f)\ f\\
&ex1 = and'\ true\ true\ True\ False \quad \text{– True}\\
&ex2 = and'\ true\ false\ True\ False \quad \text{– False}\\
&ex3 = and'\ false\ true\ True\ False \quad \text{– False}\\
&ex4 = and'\ false\ false\ True\ False \quad \text{– False}
\end{aligned}$$

and JAVASCRIPT:

```
function True(t,f) { return t; }
function False(t,f) { return f; }

function and(a,b) {
    return function (t,f) {
        return a(b(t,f),f);
    }
}

console.log(and(True,True)(true,false)); // true
console.log(and(True,False)(true,false)); // false
console.log(and(False,True)(true,false)); // false
console.log(and(False,False)(true,false)); // false
```

Here we had to "reduce" church-encoded data to the languages' primitive data, by applying primitive *true* and *false* literals. In bare λ-calculus setting, we'd use α-equivalence to determine whether the end result is *true* or *false*.

Example 2.11 (Reduction)**.** As the second last example, we'll show how a substitution evaluation model works for the *and* function defined above. It's worth noticing that arbitrary λ-calculus term reduction may not terminate.

There are two systematic reduction orders:

- *Normal order reduction*. Reduce the leftmost outermost β-redex first (call-by-need)
- *Applicative order reduction*. Reduce the leftmost innermost β-redex first (call-by-value).

First we substitute definition for their names in the expression, then try to apply principal evaluation scheme $(\lambda x \to M)\ N =_\beta [x/N]M$ (Definition 2.5) in the applicative order.

$$
\begin{aligned}
&\textit{and true false} \\
&\quad = (\lambda a\ b\ t_0\ f_0.\, a\ (b\ t_0\ f_0)\ f_0)\ (\lambda t_1\ f_1.\, t_1)\ (\lambda t_2\ f_2.\, f_2) \\
&\quad =_\beta (\lambda b\ t_0\ f_0.\, (\lambda t_1\ f_1.\, t_1)\ (b\ t_0\ f_0)\ f_0)\ (\lambda t_2\ f_2.\, f_2) \\
&\quad =_\beta (\lambda b\ t_0\ f_0.\, (\lambda f_1.\, (b\ t_0\ f_0))\ f_0)\ (\lambda t_2\ f_2.\, f_2) \\
&\quad =_\beta (\lambda b\ t_0\ f_0.\, b\ t_0\ f_0)\ (\lambda t_2\ f_2.\, f_2) \\
&\quad =_\beta \lambda t_0\ f_0.\, (\lambda t_2\ f_2.\, f_2)\ t_0\ f_0 \\
&\quad =_\beta \lambda t_0\ f_0.\, (\lambda f_2.\, f_2)\ f_0 \\
&\quad =_\beta \lambda t_0\ f_0.\, f_0 \\
&\quad =_\alpha \lambda t\ f.\, f \\
&\quad = \textit{false}
\end{aligned}
$$

And another expression shows the normal order reduction. Note how both function and argument values of some β-redexes aren't in the normal form themselves.

$$
\begin{aligned}
&\textit{and false true} \\
&\quad = (\lambda a\ b\ t_0\ f_0.\, a\ (b\ t_0\ f_0)\ f_0)\ (\lambda t_1\ f_1.\, f_1)\ (\lambda t_2\ f_2.\, f_2) \\
&\quad =_\beta (\lambda b\ t_0\ f_0.\, (\lambda t_1\ f_1.\, f_1)\ (b\ t_0\ f_0)\ f_0)\ (\lambda t_2\ f_2.\, f_2) \\
&\quad =_\beta \lambda t_0\ f_0.\, (\lambda t_1\ f_1.\, f_1)\ ((\lambda t_2\ f_2.\, f_2)\ t_0\ f_0)\ f_0 \\
&\quad =_\beta \lambda t_0\ f_0.\, (\lambda f_1.\, f_1)\ f_0 \\
&\quad =_\beta \lambda t_0\ f_0.\, f_0 \\
&\quad =_\alpha \lambda t\ f.\, f \\
&\quad = \textit{false}
\end{aligned}
$$

Example 2.12 (Y-combinator)**.** As the last example in this section we will consider Curry's paradoxical Y-combinator.

$$Y = \lambda f.\,(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$
$$Y\ f = (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

This combinator may be used in implementing Curry's paradox. [18].

As we see the combinator is not in its normal form, there are sub-terms of the form $(\lambda x.M)N$. Let's try to do a single reduction:

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

We'll see that the term is recurring:

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

Using Y-combinator we can represent general recursion, even if we cannot refer to names recursively in the λ-calculus!

Logically λ-calculus is inconsistent. We can write non-normalising terms, and we cannot decide whether terms have normal forms, because of a *halting problem*: λ-calculus is an universal language. Computationally weaker, but logically consistent versions were developed. There are various *typed* lambda calculi which are *total* i.e. their reductions will always terminate to the unique form.

From the pragmatic point of view, types would help programmers to catch many kinds of errors in the programs. Also if we have to remember types of the arguments, why not to write that down. Then types would act as machine-checked documentation. We can use a type-system to encode various properties of the programs, which hold if the program is well-typed (for example: termination or correct memory access).

The next sections introduce various typed lambda calculi in increasing order of expressiveness of the type systems, from simply typed lambda calculus to a fully dependent type system.

2.2 Simply typed lambda calculus: $\lambda^{\to}$

In this section we introduce the most elementary typed language, *simply typed lambda calculus*, STLC or $\lambda^{\to}$ [5].

There are many reasons to have a type system in the programming language: documentation, guiding code generation or ruling out various program errors. The motivation for development of $\lambda^{\rightarrow}$ was of the last type, to forbid erroneous programs, e.g. Y-combinator.

The simply typed lambda calculus is only a small syntactic extension of the untyped one. We'll need a small language for types, with primitive types and function arrow, and also constants of known types. We'll annotate lambda abstraction with the type of the variable. The syntax is shown in Figure 2.1. Usually we omit the parentheses in λ-abstraction, and write $\lambda x : \tau . e$.

TYPES	τ	$:= \alpha \in \mathbb{B}$	*basis type*
		$\mid \tau \rightarrow \tau$	*function*
TERMS	e	$:= c \in \mathbb{C}$	*constant*
		$\mid x$	*variable*
		$\mid e_1\ e_2$	*application*
		$\mid \lambda(x : \tau).e$	*abstraction*

Figure 2.1: $\lambda^{\rightarrow}$ syntax

As we can see the language is very similar to untyped lambda calculus. The typing rules are syntax directed. Given any $\lambda^{\rightarrow}$-expression, it's trivial to check whether it's well-typed.

As we can see, the syntax of terms is very close to untyped lambda calculus. In fact, the type annotation of the type of the variable in the λ-term is not necessary. Given the type context defining types of the constants, the type of expression could be easily inferred, as we will see in Chapter 3. The formulation with explicit type annotations is called *Church notation*, and the implicit one is *Curry notation*.

Definition 2.13 (Rule, Context and Judgment)**.** A *judgment* is a "meta-proposition". One writes

$$\vdash J$$

to mean that J is a judgment that is derivable, i.e. a theorem of the deductive system.

It may happen that a judgment J is only derivable under the assumptions

of certain other judgments $J_1, \ldots, J_n$. In this case one writes

$$J_1, \ldots, J_n \vdash J.$$

Hardly ever do we want to write all assumptions we have, so we combine them in a *context*, Γ.

Often, however, it is convenient to incorporate hypotheticality into judgments themselves, so that $J_1, \ldots, J_n \vdash J$ becomes a single hypothetical judgment. It can then be a consequence of other judgments, or (more importantly) a hypothesis used in concluding other judgments. For instance, in order to conclude the truth of an implication $\phi \Rightarrow \psi$, we must conclude ψ assuming ϕ; thus the introduction rule for implication is

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \ {\to}\mathrm{I}$$

Remark 2.14. When working with multiple different systems, the subscript is often used to differentiate in which system judgments are made, for example J is derivable from Γ in system A:

$$\Gamma \vdash^A J.$$

However, we often omit the subscript when it's clear from the outer context.

TYPE CONTEXT	$\Gamma ::= \varepsilon$		*empty context*
	$\mid \ \Gamma, x : \tau$		*extended context*

Figure 2.2: Type context

$$\frac{}{\Gamma \vdash c : [\![c]\!]} \ \textsc{Const} \qquad \frac{}{\Gamma, x : \alpha \vdash x : \alpha} \ \textsc{Start} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma, y : \sigma \vdash x : \tau} \ \textsc{Weaken}$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 \ e_1 : \tau'} \ \textsc{App} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'} \ \textsc{Abs}$$

Figure 2.3: $\lambda^{\to}$ type system

The type system of $\lambda^{\to}$ is presented in the Figure 2.3. The left-hand-side of the judgments (called the *context*, Figure 2.2) contains typing declarations of

variables that might occur freely on the right-hand-side. In the CONST-tule we use *typeof*-operator, $[\![\cdot]\!]$. Here it means that the type of constants has to be known a priori. Later we will talk about types of compound expressions. Alternatively we can omit the CONST rule, and provide constants in the initial environment. The START is used to conclude typing based on the right-most typing declarations in the context. To use other declarations in the context, we use WEAKEN rule to dismiss unnecessary declarations. This rules may be combined into single

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha} \text{ VAR}$$

rule, but we will use more explicit START & WEAKEN -rules. In $\lambda^{\rightarrow}$ as well as in System F covered in Section 2.3 the context is a set; the order of the judgements in the context doesn't matter, as they are all independent. However, in the Pure Type System formalisation, extending to dependent type systems (Section 2.4), context is a more complicated "dependent list".

The typing judgments feel natural, if we abstract over a variable and then apply a value, the type of expression will be the same as in the beginning. Also vice-versa, if we apply a value, and then abstract over it, the type will again be the same. One can say, that the underlying logic of the $\lambda^{\rightarrow}$ type system, the propositional logic, is *sound* and *complete*. Informally this means that the rules are correct and that no other rules are required. [1]

This language may seem very limited, but it's powerful enough to simulate C's or pre-generics Java type systems. In C there are a few polymorphic operators, like array-access brackets. Also arrays and pointers are polymorphic types. But the programmer cannot define similar polymorphic types or functions. We will cover them in Section 2.3. Actually arrays are parametrised over their size as well, which would require dependent-types Section 2.4.

```
// getSecond : Array 5 Int -> Int
// getSecond = \arr -> [] arr 2
int getSecond(int[5] arr) {
    return arr[1];
}
```

[1] For example in modal logic, defining sound and complete elimination rule for a validity operator is non-trivial[58]. Alternative definitions are also possible, giving different structure of the proofs/programs [52]

Example 2.15. $\lambda^{\rightarrow}$ with booleans Now we can type the Boolean example from the untyped lambda calculus section (Example 2.9). The simply typed lambda calculus isn't powerful enough to define booleans using Church-encoding, so the equivalent values should be in the given constants.

$$\mathbb{B} = \{Bool\}$$
$$\mathbb{C} = \{true : Bool, false : Bool, boolElim_{Bool} : Bool \rightarrow Bool \rightarrow Bool \rightarrow Bool\}$$

$$\begin{aligned} \text{TRUE-ELIM} \quad & boolElim_{Bool}\ true\ t\ f = t \\ \text{FALSE-ELIM} \quad & boolElim_{Bool}\ false\ t\ f = f \end{aligned}$$

In addition to specifying the type basis and constants, we provide reduction rules for $boolElim_{Bool}$. Without them application would be irreducible, in other words: not very useful.

The type $boolElim_{Bool}$ is confusing, but it's a well-known if-else-then operator with boolean as a result (we cannot make it generic i.e. polymorphic in $\lambda^{\rightarrow}$) Using this constant set we can define an *and* function:

$$and\ a\ b = boolElim_{Bool}\ a\ (boolElim_{Bool}\ b\ true\ false)\ false$$

Or even simpler:

$$and\ a\ b = boolElim_{Bool}\ a\ b\ false$$

All the interesting stuff is now happening inside the provided constants:

$$and\ true\ false = boolElim_{Bool}\ true\ false\ false = false$$
$$and\ false\ true = boolElim_{Bool}\ false\ true\ false = false$$

Similarly we can define other functions on booleans: *or*, *xor*, etc.

Example 2.16 (Y-combinator)**.** To perform arbitrary computations (i.e. to be Turing complete, generally useful) we need either arbitrary loops or arbitrary i.e. general recursion. Having general recursion also means that every type is inhabited by nonterminating computation. This is a compromise programming language make, so we can write more programs. However, having general recursion on the type-level is probably a bad thing, as type-checking could be nonterminating. Yet many languages with expressive meta-programming features have Turing-complete type-level language (C++ [69], SCALA [23], HASKELL with GHC extensions [22]).

Simple typed lambda calculus doesn't suffer from Curry paradox. The evaluation of $\lambda^{\to}$-terms always terminates, in other words $\lambda^{\to}$ is *total*, given functions in $\mathbb{C}$ set are total, as well. We cannot define Y-combinator, or any other infinitely-evaluating expressions. Yet we can give a type to the Y-combinator and add it to the $\mathbb{C}$ set, specialised to different types:

$$\mathit{fixBool} : (\mathit{Bool} \to \mathit{Bool}) \to \mathit{Bool}$$

whose execution is defined by

$$\mathit{fixBool}\, f = f\ (\mathit{fix}\, f)$$

And now we can perfectly write

$$\begin{aligned}&\mathit{fixBool}\ (\mathit{and}\ \mathit{false})\\ &\quad = \mathit{and}\ \mathit{false}\ (\mathit{fixBool}\ (\mathit{and}\ \mathit{false}))\\ &\quad = \mathit{false}\end{aligned}$$

However the opposite example:

$$\begin{aligned}&\mathit{fixBool}\ (\mathit{and}\ \mathit{true})\\ &\quad = \mathit{and}\ \mathit{true}\ (\mathit{fixBool}\ (\mathit{and}\ \mathit{true}))\\ &\quad = \mathit{and}\ \mathit{true}\ (\mathit{and}\ \mathit{true}\ (\mathit{fixBool}\ (\mathit{and}\ \mathit{true})))\\ &\quad = \ldots\end{aligned}$$

is not terminating. If we reason equationally:

$$\mathit{result} = \mathit{and}\ \mathit{true}\ \mathit{result}$$

then both *true* and *false* satisfy the equation!

This kind of paradoxes are unfortunate. Especially when we extend type-systems to allow computation on the type-level, it's desirable that terms have an unique type, which is also computable in a finite time.

To back a bit: why cannot we define the Y-combinator? The problem is better explained using polymorphic types, so we postpone it until Section 2.3.

For different type-systems we may ask several questions:

- Given x and σ, does one have $\vdash x : \sigma$?
- Given x, does there exists a σ such that $\vdash x : \sigma$?
- Given σ, does there exists a x such that $\vdash M : \sigma$?

These three problems are called *type-checking*, *typability* and *inhabitation* respectively. For the $\lambda^{\to}$ all three problems are decidable[2] With addition of the Y-combinator, inhabitation problem for $\lambda^{\to}$ turns into trivial one, as there is $\vdash fix_{\alpha}\ id_{\alpha} : \alpha$ term. Chapter 3 will discuss the *typability* problem.

As C-example shows, the programmers' abstraction possibilities are limited. With the introduction of separate languages for the types, we would like to have abstraction possibilities there, as well. That leads us to the polymorphic lambda calculus, also known as *System F*.

2.3 System F: $\lambda 2$

In System F, $\lambda 2$ [31] it is possible to have types universally quantify over types. As we can see from previous section, $\lambda^{\to}$ is restricted, we cannot even write generic function $boolElim$, but have to resort to different specific functions like

$$boolElim_{Bool} : Bool \to Bool \to Bool \to Bool,$$
$$boolElim_{Int} : Bool \to Int \to Int \to Int.$$

But with lambda abstraction over types we could have:

$$boolElim : \forall\alpha.\ Bool \to \alpha \to \alpha \to \alpha.$$

Such addition to expressiveness comes with the cost, System F types aren't inferable. Even the type-checking of Curry-style System F isn't decidable, as it's equivalent to type-inference [72]. The type-checking of Church-style is decidable, as all needed information is present in the annotations.

The syntax of System F language is shown in Figure 2.4. The type system is presented in Figure 2.5. The System F is a strict superset of $\lambda^{\to}$, which means that every simple type or simply typed expression is a valid System F type or expression respectively. The type basis and constants are omitted from above definitions, as they aren't strictly necessary to write programs in System F. Basic types like booleans could be defined using Church-encoding, much like in untyped lambda calculus case. [45][3] We can see it from the typing rules, we added instantiation, INST and generalisation GEN

[2]For System F (Section 2.3), inhabitation is undecidable, as well as typability and type-checking of Curry-style System F according to Wells [72]. Church-style systems obviously have decidable type-checking.

[3]Leivant defines datatypes directly in $\lambda 2$.

TYPES	τ	$:= \tau \to \tau$	*function*
		$\mid \forall\alpha.\,\tau$	*type abstraction, polymorphic type*
TERMS	e	$:= x$	*variable*
		$\mid e_1\ e_2$	*application*
		$\mid \lambda(x:\tau).\,e$	*abstraction*
		$\mid e\ \tau$	*type application*
		$\mid \Lambda\alpha.\,e$	*type abstraction*

Figure 2.4: System F syntax

$$\frac{}{\Gamma, x:\alpha \vdash x:\alpha}\ \textsc{Start} \qquad \frac{\Gamma \vdash x:\tau}{\Gamma, y:\tau' \vdash x:\tau}\ \textsc{Weaken}$$

$$\frac{\Gamma \vdash e_0:\tau \to \tau' \quad \Gamma \vdash e_1:\tau}{\Gamma \vdash e_0\ e_1:\tau'}\ \textsc{App} \qquad \frac{\Gamma, x:\tau \vdash e:\tau'}{\Gamma \vdash (\lambda x:\tau.\,e):\tau \to \tau'}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash x:\forall\alpha.\,\tau}{\Gamma \vdash x\ \sigma:[\alpha/\sigma]\ \tau}\ \textsc{Inst} \qquad \frac{\Gamma \vdash x:\tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha.\,x:\forall\alpha.\,\tau}\ \textsc{Gen}$$

Figure 2.5: System F type system

rules to deal with polymorphism. Especially the instantiation rule might look weird, as we "apply" type σ to the expression, but that's exactly what for example recent `TypeApplications` GHC HASKELL extension allows us to do.

Example 2.17 (Booleans)**.** We mentioned above that Church-encoded booleans are possible in System F. The only difference is that we must have explicit type abstraction:

$$\begin{aligned} \textbf{type}\ \mathit{Bool} &= \forall\alpha.\,\alpha \to \alpha \to \alpha \\ \mathit{true} &= \Lambda\alpha.\,\lambda t : \alpha.\,\lambda f : \alpha.\,t \\ \mathit{false} &= \Lambda\alpha.\,\lambda t : \alpha.\,\lambda f : \alpha.\,f \end{aligned}$$

Let us type-check the expression of *true*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t : \alpha \vdash t : \alpha}\ \textsc{Start}}{t : \alpha, f : \alpha \vdash t : \alpha}\ \textsc{Weaken}}{t : \alpha \vdash \lambda f : \alpha.\,t : \alpha \to \alpha}\ \textsc{Abs}}{\varepsilon \vdash \lambda t : \alpha.\,\lambda f : \alpha.\,t : \alpha \to \alpha \to \alpha}\ \textsc{Abs}}{\varepsilon \vdash \Lambda a.\,\lambda t : \alpha.\,\lambda f : \alpha.\,t : \forall\alpha.\,\alpha \to \alpha \to \alpha}\ \textsc{Gen}$$

Using the above definitions we can once again define an *and* function:

$$\begin{aligned} &\mathit{and} : \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\ &\mathit{and} = \lambda x : \mathit{Bool}.\,\lambda y : \mathit{Bool}.\,x\ \mathit{Bool}\ y\ \mathit{false}. \end{aligned}$$

We can typecheck this example as well, see the derivation in Appendix A.1. As the term is well-typed, we can erase type annotations and abstractions, and evaluate the term exactly as in the untyped lambda calculus case.

Example 2.18 (Y-combinator)**.** Continuing from $\lambda^{\to}$ Y-combinator example. Now we could give a type to a generic fixed-point operator: $\mathit{fix} : (\alpha \to \alpha) \to \alpha$. Yet we cannot fill the missing types in the Y-combinator expression, give the types to subexpressions: $Y = \lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))$ The problematic part is self-application: $x\ x$. Then only applicable typing judgement is APP:

$$\dfrac{\Gamma \vdash x : \beta \to \alpha \qquad \Gamma \vdash x : \beta}{\Gamma \vdash x\ x : \alpha}\ \textsc{App}$$

From which we get that the type of x should be both $\beta \to \alpha$ and β, which would require recursive type definition:

$$\textbf{type}\ \beta = \beta \to \alpha$$

Such types cannot be expressed in System F. In a type-system with equi-recursive types, such type could be written as

$$\textbf{type}\ \beta = \mu\gamma.\ \gamma \to \alpha$$

The equi-recursive style places stronger demands on the typechecker, which must work with infinite structures. Moreover, the interactions between equi-recursive types and other advanced typing features can be quite complex, leading to significant theoretical difficulties. [59]

Probably for this reasons Haskell uses iso-recursive types, i.e. requiring explicit constructor or **newtype** wrapper:

$$\textbf{newtype}\ Nu\ \alpha = Nu\ (Nu\ \alpha \to \alpha)$$

Yet for example AGDA and COQ forbids such recursive types[4]. Types like *Nu* would make every type inhabitant, thus making underlying logic unsound, see Appendix B.1 for Agda example.

Another interesting property of System F is *impredicativity*. A type system that allows a polymorphic function to be instantiated at a polymorphic type is called *impredicative*, while a predicative system only allows a polymorphic function to be instantiated with a monomorphic type. Generally, self-referencing definition is called impredicative.

Definition 2.19 (Impredicativity)**.** More precisely, a definition is said to be *impredicative* if it invokes (mentions or quantifies over) the set being defined, or (more commonly) another set which contains the thing being defined.

Type-inference is much easier in a predicative type systems. In Section 3.3 we will discuss a restriction of System F, which is predicative.

In this section we introduced polymorphic lambda calculus: System F. It's already a very expressive calculus, which is a basis for modern functional languages like Haskell or ML. Yet (as always?) this is not the end of story. More features can be added to the type-systems.

[4]Recursion can occur only in strictly positive position, loosely speaking "on the right side of the arrow". for example **data** $Nat = Z \mid S\ Nat$

2.4 Dependent type systems

The natural course of the type-system development is to introduce even more abstraction mechanisms, making type-system more expressive or/and robust. There are various ways to arrive there, and we will go through an a bit artificial path by improving upon System F in the right direction.

2.4.1 System F as a Pure Type System

The method of generating the systems in the λ-cube has been generalized independently by Berardi and Terlouw [7, 66]. This resulted in the notion of *pure type system* PTS. Many systems of typed lambda calculus a lá Church can be seen as PTSs. Subtle differences between systems can be described neatly using the notation for PTSs. [5]

If we look at the type-checking judgment of *true* in Example 2.17, the type variables occur out of nothing. Also we have unelegant $\alpha \notin FV(\Gamma)$ -check. The Γ context contains only the term-variables (we can interpret other symbols as constants). There isn't similar context for type-variables. We can introduce a kind context, which will change e.g. GEN rule to:

$$\frac{\Gamma \vdash x : \tau \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha.\, x : \forall\alpha.\, \tau}\ \textsc{Gen} \quad \longmapsto \quad \frac{\Delta, \alpha; \Gamma_{\Delta,\alpha} \vdash x : \tau}{\Delta; \Gamma_{\Delta} \vdash \Lambda\alpha.\, x : \forall\alpha.\, \tau}\ \textsc{Gen}$$

Now the type context Γ is parametrised by the kind context. Δ contains all type-variables which can be used in the types. We could add similar context, with quite simple rules to check well-scopedness (and closedness) of untyped lambda calculus terms. Trying to mechanise this approach[5]you may notice that the type language resembles $\lambda^{\to}$ very much (compare the new GEN rule to the ABS rule), the type of types is *kinds*. You might ask, what if we would have System F on type level, what will the term language look like, i.e. what if we will have type of kinds.

There are different ways to make that construction elegantly, where the simple, yet non-obvious one is to unify types and terms. We will briefly introduce *Pure type systems*, describe System F in the new formalism, and later generate other, more expressive type systems as PTS.

Definition 2.20. The *specification* of a PTS consists of a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

[5]See `https://gist.github.com/phadej/780c1f5706b6cee805bd` for System F and `https://gist.github.com/phadej/82084de18b314701fa7e` for $\lambda^{\to}$. Also Appendix B.5, TYPESSON also has kinds, which are used to construct well-typed type context.

- $\mathcal{S}$ is subset of constants, called *sorts*.
- $\mathcal{A}$ is a set of *axioms* of the form $c : s$ where $c \in \mathcal{C}$ and $s \in \mathcal{S}$.
- $\mathcal{R}$ is a set of *rules* of the form (s_1, s_2, s_3) with $s_1, s_2, s_3 \in \mathcal{S}$.

See Figure 2.6 and Figure 2.7 for the syntax and the typing rules respectively.

PTS-TERMS	$t :=$	x	*variable*
	$\mid$	c	*constant*
	$\mid$	$\lambda(x : t).\, t$	*abstraction*
	$\mid$	$\Pi(x : t).\, t$	Π*-type, dependent product*
	$\mid$	$t\ t'$	*application*

Figure 2.6: Pure type systems' syntax

$$\frac{c : s \in \mathcal{A}}{\varepsilon \vdash c : s}\ \textsc{Axiom} \qquad \frac{}{\Gamma, x : \alpha \vdash x : \alpha}\ \textsc{Start}$$

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash \beta : s \quad s \in \mathcal{S}}{\Gamma, y : \beta \vdash x : \alpha}\ \textsc{Weakening}$$

$$\frac{\Gamma \vdash \alpha : s_1 \quad \Gamma, x : \alpha \vdash \beta : s_2 \quad s_1, s_2, s_3 \in \mathcal{R}}{\Gamma \vdash (\Pi x : \alpha.\beta) : s_3}\ \textsc{Product}$$

$$\frac{\Gamma \vdash f : (\Pi x : \alpha.\beta) \quad \Gamma \vdash y : \alpha}{\Gamma \vdash f\ x : [x/y]\ \beta}\ \textsc{Application}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta \quad \Gamma \vdash (\Pi x : \alpha.\, \beta) : s \quad s \in \mathcal{S}}{\Gamma \vdash (\lambda x : \alpha.\, e) : (\Pi x : \alpha.\, \beta)}\ \textsc{Abstraction}$$

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash \beta : s \quad \alpha =_\beta \beta \quad s \in \mathcal{S}}{\Gamma \vdash x : \beta}\ \textsc{Conversion}$$

Figure 2.7: Pure type system

The rules probably need some explanation. The AXIOM, START, and WEAKENING rules are straight-forward. The PRODUCT rule is about dependent-product. The non-dependent product is an ordinary function, but in the dependent variant, the type of the result may depend on the value of the

argument. The APPLICATION and ABSTRACTION work with dependent products. See also Remark 2.22. The last rule, CONVERSION is needed because to compare types, we might need to compute some beta-reductions.

Example 2.21 (System F as PTS)**.** For expressing System F we need only two constants $\star$ and $\Box$.

$$\begin{array}{ll} \mathcal{S} & \star, \Box \\ \mathcal{A} & \star : \Box \\ \mathcal{R} & (\star, \star, \star), (\Box, \star, \star) \end{array}$$

Judgment $\tau : \star$ can be read as "τ is a type". Respectively, $\star : \Box$ can be read as "$\star$ is a kind". There are no other kinds then $\star$ n System F, but later we want to talk about type constructors which have kind $\star \to \star$ etc.[6] Many type-systems can be described by picking different SAR-triples, for example $\lambda^{\to}$ is almost the same, except $\mathcal{R}$ has only $(\star, \star, \star)$-rule.

As there are two triples in $\mathcal{R}$, there are also two (dependent-)products, i.e. what we can abstract over: universal quantification ($\forall$) and functions ($\to$). Thus Π generalises both:

$$\begin{aligned} \alpha \to \beta &\equiv \Pi x : \alpha . \beta \\ \forall \alpha . \beta &\equiv \Pi \alpha : \star . \beta \end{aligned}$$

Respectively we use λ for both Λ and λ of System F.

Remark 2.22. The rules $\mathcal{R}$ define which kind of (dependent-) products we can make, and they determine which abstractions we can make. We can fuse PRODUCT and ABSTRACTION into one, which may help understand the idea:

$$\dfrac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\lambda x : \alpha . e) : (\Pi x : \alpha . \beta)} \text{ ABSTRACTION} \qquad \dfrac{\Gamma \vdash \alpha : s_1 \qquad \Gamma, x : \alpha \vdash \beta : s_2}{\Gamma \vdash (\Pi x : \alpha . \beta) : s_3} \text{ PRODUCT}$$

into

$$\dfrac{\Gamma \vdash \alpha : s_1 \qquad \Gamma, x : \alpha \vdash e : \beta : s_2}{\Gamma \vdash (\lambda x : \alpha . e) : (\Pi x : \alpha . \beta) : s_3} \text{ PRODABS}$$

[6] We can have type *List* α, but we cannot talk about non-applied *List*, we need $(\Box, \Box, \Box)$ -rule for that. Adding it will give us a $\lambda\omega$ system.

Here we use a shorthand notation: $a : b : c$ to mean $a : b, b : c$. Let's take for example $(\Box, \star, \star)$ rule of System F, note how $\star : \Box$ axiom is essential here as well.

$$\frac{\Gamma \vdash \star : \Box \qquad \Gamma, x : \star \vdash e : \beta : \star}{\Gamma \vdash (\lambda x : \star . e) : (\Pi x : \star . \beta) : \star} \text{ PRODABS-}\Box, \star, \star$$

which is the same as the GEN rule in Figure 2.5.

Let us revisit booleans once again:

$$\begin{aligned} \mathit{Bool} &= \Pi\alpha : \star . \Pi t : \alpha . \Pi f : \alpha . \alpha \\ \mathit{true} &= \lambda\alpha : \star . \lambda t : \alpha . \lambda f : \alpha . t \\ \mathit{false} &= \lambda\alpha : \star . \lambda t : \alpha . \lambda f : \alpha . f \end{aligned}$$

First we can show that *Bool* is a type, i.e. $\varepsilon \vdash \mathit{Bool} : \star$

$$\dfrac{\dfrac{\varepsilon \vdash \star : \Box \qquad \dfrac{\alpha : \star \vdash \Pi t : \alpha . \Pi f : \alpha . \alpha : \star}{\varepsilon \vdash \Pi\alpha : \star . \Pi t : \alpha . \Pi f : \alpha . \alpha : \star} \qquad \dfrac{\alpha : \star \vdash \alpha : \star \qquad \dfrac{\dfrac{\alpha : \star \vdash \alpha : \star}{\alpha : \star, t : \alpha \vdash \alpha : \star} \qquad \dfrac{\alpha : \star \vdash \alpha : \star}{\alpha : \star, t : \alpha, f : \alpha \vdash \alpha : \star}}{\alpha : \star, t : \alpha \vdash \Pi f : \alpha . \alpha : \star}}{\alpha : \star \vdash \Pi t : \alpha . \Pi f : \alpha . \alpha : \star}}$$

And now we can show that $\varepsilon \vdash \mathit{true} : \mathit{Bool}$, we omit the type derivation of type-part of ABSTRACTION rule, as they are the same as above (the first is $\varepsilon \vdash \mathit{Bool} : \star$). The complete derivation is shown in Equation (A.3).

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha : \star, t : \alpha \vdash t : \alpha}{\alpha : \star, t : \alpha, f : \alpha \vdash t : \alpha}}{\alpha : \star, t : \alpha \vdash \lambda f : \alpha . t : \Pi f : \alpha . \alpha}}{\alpha : \star \vdash \lambda t : \alpha . \lambda f : \alpha . t : \Pi t : \alpha . \Pi f : \alpha . \alpha} \qquad \dfrac{\vdots}{\varepsilon \vdash \mathit{Bool} : \star}}{\varepsilon \vdash \lambda\alpha : \star . \lambda t : \alpha . \lambda f : \alpha . t : \Pi\alpha : \star . \Pi t : \alpha . \Pi f : \alpha . \alpha}$$

Using the PTS formalism we can easily define new type-systems, by using different $\mathcal{S}, \mathcal{A}, \mathcal{R}$ triples. With only a single rule: $\mathcal{R} = (\star, \star, \star)$ we get $\lambda^{\rightarrow}$, which easily shows that $\lambda^{\rightarrow}$ is a subsystem of System F. By using different rules we get other type systems, we will explore some of them in the next sections.

2.4.2 System $\lambda\star$

This subsection is a slight detour, to address one innocent looking attempt to dependent-types. The system $\lambda\star$ in which $\star$ is the sort of all types, including itself, is specified below.

Definition 2.23 (System $\lambda\star$)**.**

$$\begin{array}{ll} \mathcal{S} & \star \\ \mathcal{A} & \star : \star \\ \mathcal{R} & (\star, \star, \star) \end{array}$$

All construction possible in λC (which is a fully dependent type system, Section 2.4.3) and also System F, can be done in $\lambda\star$ by collapsing $\Box$ to $\star$. However, the system $\lambda\star$ turns to be inconsistent in the sense that every type is inhabited.

The system $\lambda\star$ is also impredicative, as we have type in type. This is different from *impredicative polymorphism* of System F discussed previously.

Recent development in Haskell introduced $\star : \star$ rule, via `TypeInType` extension. This rule often causes type checking to be undecidable in dependently typed language [4, 12]. This axiom permits the expression of divergent terms - if the type checker tries to reduce them it will loop. However, these flaws do not concern us in Haskell. Adding dependent-type features to Haskell is not an attempt to make GHC a proof assistant. All types are already inhabited, by undefined at the term level, and by the open **type family** *Any* $(k :: \star) :: k$ at the type level. If having $\star :: \star$ allows us another way to inhabit a type, it does not change the properties of the language. [25, 71].

The $\star : \star$ axiom, might be a reasonable and simplifying choice in a language where the type-terms lack normal forms already. On the other hand, universe hierarchy in CAYENNE [4] is justified by two reasons: first, $\star : \star$ would make a type-system unsound as a logic even in the absence of recursion; second, it would make type-erasure impossible.

The inconsistency following from $\star : \star$ was first proved by Girard [31], He also showed that the circularity of $\star : \star$ is not necessary to derive the paradox. For this purpose he introduced system λU. A smaller system λU^- is also shown to be inconsistent [40]. The corollary of these proofs is that $\lambda\star$ is also inconsistent, by applying a contraction $f(\star) = f(\Box) = f(\triangle) = \star$ mapping λU and λU^- onto $\lambda\star$.

2.4.3 Calculus of constructions: λC

It's very natural to ask whether System F could be extended further. Using PTS framework it's surprisingly simple. By adding a few rules we get Calculus of Constructions, λC.

One such extension is to allow some computations on the type-level. One quite natural way is to fuse terms and types, which will result into Calculus of constructions [17].

Definition 2.24 (Calculus of constructions as PTS)**.**

$$\begin{array}{ll} \mathcal{S} & \star, \Box \\ \mathcal{A} & \star : \Box \\ \mathcal{R} & (\star,\star,\star), (\Box,\star,\star), (\star,\Box,\Box), (\Box,\Box,\Box) \end{array}$$

There are two new rules in addition to the rules in System F (Example 2.21): $(\Box, \Box, \Box)$ allows us to abstract over types to produce new types, in other words write type-level functions. The $(\star, \Box, \Box)$ is a "fully-dependent-types" rule making possible types to depend on terms.

Example 2.25 (Length-indexed list)**.** The most used example of dependent types is the length-indexed vector. As we are still using church encodings to represent data types, length-indexed vectors are represented similarly as tuples. We use a shorthand notation: $\alpha \to b$ for $\Pi(x : a).\, b$.

$$\begin{aligned}
&\star^\star : \Box \\
&\star^\star = \star \to \star \\
&\mathbb{N}^{\star^\star} : \Box \\
&\mathbb{N}^{\star^\star} = \star^\star \to (\star^\star \to \star^\star) \to \star^\star \\
&\mathit{Unit} : \star^\star \\
&\mathit{Unit} = \lambda(r : \star).\, r \to r \\
&\mathit{Vector} : \alpha \to \mathbb{N}^{\star^\star} \to \star \\
&\mathit{Vector} = \lambda(\alpha : \star).\, \lambda(n : \mathbb{N}^{\star^\star}).\, \Pi(r : \star). \\
&\quad n\ \mathit{Unit}\ (\lambda(p : \star^\star).\, \lambda(r' : \star).\, \alpha \to p\ r')\ r \\
&\mathit{Vector}\ \alpha\ 0 = \Pi(r : \star).\, r \to r \\
&\mathit{Vector}\ \alpha\ 1 = \Pi(r : \star).\, \alpha \to r \to r \\
&\mathit{Vector}\ \alpha\ 2 = \Pi(r : \star).\, \alpha \to \alpha \to r \to r
\end{aligned}$$

And while λC is a very expressive type-system, there are theorems which cannot be proved in it. In other words (Curry-Howard) there are types for which terms cannot be constructed.

One very simple example is induction over natural numbers. The non-dependent induction is trivial, as it's the church encoding of natural numbers:

$$\lambda(a : \star).\, a \to (a \to a) \to a$$

Yet we will need dependent-induction to prove even simple theorems about natural numbers, which are needed for practical programming, not only for making mathematical proofs.

$$\Pi(P : \mathbb{N} \to \star).\, P\ 0 \to (\Pi(n : \mathbb{N}).\, P\ n \to P\ (succ\ n)) \to (\Pi(n : \mathbb{N}).\, P\ n)$$

We cannot derive a term with this type [30].

The above is the reason why length-indexed list example uses $\mathbb{N}^{\star^\star}$ type, and not ordinary Church encoded $\mathbb{N} = \Pi(\alpha : \star).\, \alpha \to (\alpha \to \alpha) \to \alpha$. Even if we had $Vector\ a : \mathbb{N} \to \star$, we'd need dependent induction to define simple functions such as $replicate : \Pi(a : \star)\ (n : \mathbb{N}).\, a \to Vector\ n$.

To resolve this issue inductive types were introduced to the type-theories. Definition of inductive types adds the elimination function (induction principle). It turns out that adding inductive types to λC made it inconsistent. So Calculus of constructions was also made into a predicative system, resulting into *predicative Calculus of inductive constructions*, pCIC, which is used as core of Coq proof-assistant.

The universe hierarchy (PTS rules and axioms„relationship of sorts) of the system pCIC is complicated. In next subsection we introduce one version of *Intuitionistic type theory* which in our opinion is more elegant.

2.4.4 Intuitionistic type theory

The *Intuitionistic type theory*, is another dependent type-theory. It was introduced by Per Martin-Löf [48, 49]. Instead of two sorts $\star$ and $\Box$, we have a tower of universes: $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$

Definition 2.26 (ITT as PTS)**.**

$$\begin{array}{ll} \mathcal{S} & \mathcal{U}_n, n \in \mathbb{N} \\ \mathcal{A} & \mathcal{U}_n : \mathcal{U}_{n+1} \\ \mathcal{R} & (\mathcal{U}_n, \mathcal{U}_m, \mathcal{U}_{n \sqcup m}), n, m \in \mathbb{N} \end{array}$$

where $\sqcup$ is a max function.

Proof-assistant AGDA [54] uses this form of Intuitionistic type theory, we can see it from a small check.

$$
\begin{aligned}
&f \;:\; \forall\, \{n\ m\} \to (a \;:\; \mathit{Set}\ n) \to (b \;:\; a \to \mathit{Set}\ m) \to \mathit{Set}\ (n \sqcup m)\\
&f\ a\ b \;=\; (x \;:\; a) \to b\ x
\end{aligned}
$$

Remark 2.27 (System F isn't a subsystem of ITT)**.** As ITT isn't impredicative, it's trivial to see (or try) that even church encoding of natural numbers is not typable in ITT as we want it:

$$
\begin{aligned}
&\mathit{Nat} \;:\; \mathit{Set}_1 \quad \text{– Cannot make Set}\\
&\mathit{Nat} \;=\; (a \;:\; \mathit{Set}) \to a \to (a \to a) \to a
\end{aligned}
$$

But we can define an inductive data type, and its elimination and dependent induction:

$$
\begin{aligned}
&\textbf{data}\ \mathbb{N} \;:\; \mathit{Set}\ \textbf{where}\\
&\quad \mathit{zero} \;:\; \mathbb{N}\\
&\quad \mathit{succ} \;:\; \mathbb{N} \to \mathbb{N}\\
&\mathbb{N}\text{-ind} \;:\; \{\ell \;:\; \_\} \to \mathbb{N} \to (a \;:\; \mathit{Set}\ \ell) \to a \to (a \to a) \to a\\
&\mathbb{N}\text{-ind}\ \mathit{zero}\ \ \ a\ z\ s \;=\; z\\
&\mathbb{N}\text{-ind}\ (\mathit{succ}\ n)\ a\ z\ s \;=\; s\ (\mathbb{N}\text{-ind}\ n\ a\ z\ s)\\
&\mathbb{N}\text{-dep-ind} \;:\; \{\ell \;:\; \_\}\\
&\quad \to (P \;:\; \mathbb{N} \to \mathit{Set}\ \ell)\\
&\quad \to P\ \mathit{zero}\\
&\quad \to ((n \;:\; \mathbb{N}) \to P\ n \to P\ (\mathit{succ}\ n))\\
&\quad \to ((n \;:\; \mathbb{N}) \to P\ n)\\
&\mathbb{N}\text{-dep-ind}\ P\ P0\ PS\ \mathit{zero}\ \ \ \ \ \ \;=\; P0\\
&\mathbb{N}\text{-dep-ind}\ P\ P0\ PS\ (\mathit{succ}\ n) \;=\; PS\ n\ (\mathbb{N}\text{-dep-ind}\ P\ P0\ PS\ n)
\end{aligned}
$$

The ℓ argument used is a *universe level*. Definitions above are *universe polymorphic*, which is a way to work around repeating definitions on the various levels.

The language with very expressive type-system can and is used as formal proof assistant. For many practical day-to-day programming applications such power is not necessary, we want to talk about lists of specific length,

or work with heterogeneous lists, and that might be enough. However there is some joy in being able to specify, write down the program and prove that it satisfies the given specification [53].

We would like to have expressiveness of a fully dependent type system but convenience of some scripting language. We will use AGDA as our "target" language, elaborating types from convenient source languages into AGDA. This is an instance of *universe pattern* described in the next section.

2.5 Universe pattern

As we have seen, a dependently typed language allows programmers to write functions that compute types from ordinary data. We say that the data is *a code* for the resulting types, and that the collection of types selected by the codes is a closed universe. Universes can be used for embedded domain-specific languages, to do safe ad-hoc polymorphism, and to write generic libraries. We use this approach to implement and reason about domain-specific type-systems.

We start with an untyped representation of the program, $x_{untyped}$. We elaborate this representation into a typed variant, x_{typed} which has type τ_{typed}. To see that our domain-specific language with its type-system is reasonable, we have an embedding into a bigger language: x_{host} with type τ_{host}.

The intermediate stop might seem unnecessary, but that intermediate language may have properties, such being very specific (so there aren't accidentally valid programs) and have decidable type-inference. If we can define all maps (arrows in the figure), and show that they commute, then with a bit of hand-waving we can prove that our domain-specific type-system is consistent.

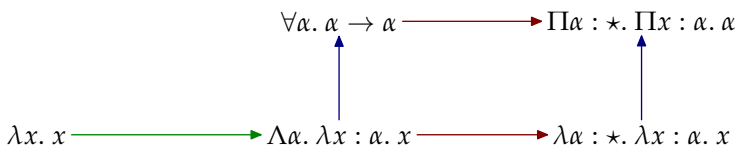

Figure 2.8: Universe pattern for untyped lambda calculus, HM and λC

Example 2.28 (λ-calculi: untyped, Hindley-Milner , and System F)**.** We can have untyped lambda calculi as an untyped language, HM as the typed

version of it, and System F as a bigger language where HM is embedded. The term and type embedding maps are trivial identity maps. We can also have Calculus of constructions in place of HM, in Figure 2.8 these relations are shown as a diagram: Hindley-Milnder terms are written with explicit types.

By proceeding through Hindley-Milner type-system we reject many programs which could be valid in System F or Calculus of Constructions. On the other hand, $\lambda^{\rightarrow}$ has decidable type-inference, as we will see later.

Example 2.29 ($\lambda^{\rightarrow}$ embedded into Agda)**.** We can also embed $\lambda^{\rightarrow}$ into Agda. We omit the representation of and embedding map of terms. The type encoding, with only natural numbers on the other hand is simple[7]. This calculus could be easily extended with e.g. type-level natural numbers (as in TypedLab Chapter 4,), which wouldn't be embeddable into System F anymore.

```
module StlcSmall where
data StlcType : Set where
  nat : Type
  _=>_ : (a b : Type) → Type
{- Embed stlc type into agda type -}
El : StlcType → Set
El nat = N
El (a ⇒ b) = El a → El b
```

For bigger example see Appendix B.2.

2.6 Conclusion

In this chapter we introduced various lambda calculi, and seen different type-systems. In many occasions we have mentioned that not all type-annotations are necessary, in many cases types can be inferred for us. We will investigate type-inference in the next chapter. Also after discussing dependent types, we shortly mentioned the universe pattern, which allows us to concentrate only on the inference of domain specific types, and get consistency of type-system for free. A larger example is TYPEDLAB which we cover in Chapter 4.

[7] thus the name: simply typed lambda calculus

Chapter 3

Type-inference

In many programming languages with a type system, also like calculi in the previous chapter, it is necessary to manually annotate symbols and functions with their types. It is then the task of the compiler to verify that these types are consistent, in other words type-check the program.

In this chapter we discuss constraint based type-inference process. This process has two distinct phases

1. the generation of constraints in the abstract syntax tree, and
2. the solving of the constraints.

The approach is the same as in HELIUM [35].

In sections 3.1 and 3.2 we use $\lambda^{\rightarrow}$ as an example language. After that we discuss Hindley-Milner type-system in Section 3.3. We conclude the chapter with generalisation of the constraint based approach in Section 3.4.

3.1 Type constraints of $\lambda^{\rightarrow}$

In our description of simply typed lambda calculus, we have explicit type annotations of the lambda abstraction variable. However, they are not strictly necessary. We can *infer* missing type annotations, if the expression can be well-typed in the first place. Similarly, we could reuse the name of the function (overload it), and a compiler can pick the right variant based on the context. There could be various *omittable* or *implicit* parts in the language, which the compiler fills in for us.

Let us concentrate on $\lambda^{\rightarrow}$ for a moment. It is quite unnecessary to write type annotations for parameters in

$$plus3 = \lambda(x : \mathbb{N})\ (y : \mathbb{N})\ (z : \mathbb{N}).\, plus\ x\ (plus\ y\ z)$$

Especially if we know that the *plus* function works only on the type *int*!

The inference algorithms in this thesis are based on the same and simple principle: gather constraints, and then solve the resulting constraint problem.

In the simply typed language case we need only one constraint type: type equality $\alpha \equiv \beta$. The language of constraints is derivable from the typing rules of the system. Recall typing rules from Figure 2.3. The rules are syntax-directed, which means that for each syntactical construct, only one rule is applicable. Each rule also gives rise to constraints, which can be seen Figure 3.1. The original typing rules are on the left. For each term where we could apply a rule, we generate a constraints shown on the right. It's worth noticing that only a single rule is applicable for each syntactic construct. This is important, as we'll see later. We denote fresh type variables with ρ letter, and use *typeof*-operator $[\![\cdot]\!]$ also to denote the type-variable used to hold the type of the expression.

$$\frac{}{\Gamma \vdash c : [\![c]\!]}\ \textsc{Const} \qquad \begin{aligned} [\![c]\!] &\equiv \rho \\ \rho &\equiv [\![c]\!] \end{aligned}$$

$$\frac{x : \alpha \in \Gamma}{\Gamma, x : \alpha}\ \textsc{Var} \qquad \begin{aligned} [\![x]\!]_{\Gamma, x:\alpha} &\equiv \rho \\ \rho &\equiv [\![\alpha]\!] \end{aligned}$$

$$\frac{\Gamma \vdash f : \tau \rightarrow \tau' \quad \Gamma \vdash x : \tau}{\Gamma \vdash f\ x : \tau'}\ \textsc{App} \qquad \begin{aligned} [\![f\ x]\!]_{\Gamma} &\equiv \rho \\ [\![f]\!]_{\Gamma} &\equiv [\![x]\!]_{\Gamma} \rightarrow \rho \end{aligned}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \rightarrow \tau'}\ \textsc{Abs} \qquad \begin{aligned} [\![\lambda x.\, e]\!]_{\Gamma} &\equiv \rho \\ \rho &\equiv \rho' \rightarrow [\![e]\!]_{\Gamma, x:\rho'} \end{aligned}$$

Figure 3.1: Constraint generation for $\lambda^{\rightarrow}$

If we omit the type annotations in the above example:

$$plus3 = \lambda x\ y\ z.\, plus\ x\ (plus\ y\ z)$$

We can gather constrains easily by traversing the syntactical structure of the expression. We introduce a new type variable for each expression, and using the information about types of subexpressions we can tell the constraints.

Example 3.1 (Collecting constraints for $f\ x$)**.** For example the application $f\ x$. The part of algorithm of constraint gathering can be written in Haskell as:

```
collect :: ConstraintMonad m
   ⇒ [TypeVar]     – Type of bounded variables
   → Term          – Term to type
   → m TypeVar     – Type variable for the term type
collect ctx (App f x) = do
  fType ← collect ctx f
  xType ← collect ctx x
  yType ← newVariable
    – fType ≡ xType → yType
  tellConstraint (typeEqual fType (typeArrow xType yType))
  pure yType
```

The *ConstraintMonad* above offers two actions: generating fresh constraint variables (*State*) and ability to gather generated constraints (*Writer*).

```
class ConstraintMonad m where
  newVariable :: m TypeVar
  tellConstraint :: TypeConstraint → m ()
```

Example 3.2 (Constraints of *plus3*)**.** When we collect constraints from the *plus*3 example above, we get the following constraints:

$$
\begin{array}{lll}
[\![plus3]\!] = 11 & [\![plus]\!] = 0 & [\![plus]\!] = 3 \\
[\![x]\!] = 1 & [\![y]\!] = 4 & [\![z]\!] = 6 \\
0 \equiv \mathbb{N} \to \mathbb{N} \to \mathbb{N} & 0 \equiv 1 \to 2 & 3 \equiv \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
3 \equiv 4 \to 5 & 5 \equiv 6 \to 7 & 2 \equiv 7 \to 8 \\
9 \equiv 6 \to 8 & 10 \equiv 4 \to 9 & 11 \equiv 1 \to 10
\end{array}
$$

We omit the type-variables of the subexpressions, to one more:: the body of the function is variable $[\![plus\ x\ (plus\ y\ z)]\!] = 8$.

Formally, to support type annotations, we'll need to add a new typing rule

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash (x : \tau) : \tau}\ \textsc{Annot}.$$

At this point we'll relax our formalism, and implicitly include this rule to all type-systems in this work.

Example 3.3 (Partial typing)**.** In this approach it's easy to add full or partial type annotations, as those would result as additional constraints. For example we could have a partially typed program

$$plus3 = \lambda(x : \mathbb{N})\ y\ z.\, plus\ x\ (plus\ y\ z)$$

In addition to the constraints from the previous example, we'll only need to add $1 \equiv \mathbb{N}$ constraint, as we "know" what type of x is.

Alternatively we could write type signature to the whole function, using underscore for omitted parts of the signature:

$$plus3 : \mathbb{N} \to \_ \to \_ \to \_$$
$$plus3 = \lambda x\ y\ z.\, plus\ x\ (plus\ y\ z)$$

In this case also a single additional constraint would be sufficient: $11 \equiv \mathbb{N} \to 12 \to 13 \to 14$

After we have gathered constraints, we have to actually find a solution to the constraint solving problem. In the following section, we'll discuss *unification*, as a method to solve equality constraints.

3.2 Unification

After the constraints are collected, we need a method to solve them. Structural equality constraints can be solved via *unification*. The solution of a unification problem is a substitution, that is, the function from problem's variables to the expressions.

For example, using x, y, z as variables, the singleton equation set

$$cons\ x\ (cons\ x\ nil) = cons\ 2\ y$$

is a syntactic first-order unification problem that has the substitution

$$x \longmapsto 2$$
$$y \longmapsto cons\ 2\ nil$$

as its only solution. The syntactic first-order unification problem

$$y = cons\ 2\ y$$

has no solution over the set of finite terms that would result into construction of infinite types, which we disallow. Otherwise Y-combinator would be typable (see Example 2.12, Example 2.16 and Example 2.18).

The first algorithm is given by Robinson [64] yet more efficient algorithms are developed [47]. From our perspective, the details of the underlying algorithm are not important. The library we use, `unification-fd`, has powerful abstraction: Unification monad, which frees us from nasty details of the low-level implementation.

Example 3.4. The idea of unification is simple, let's take two first constraints from Example 3.2:

$$\begin{aligned} 0 &\equiv \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ 0 &\equiv 1 \to 2. \end{aligned}$$

It's quite obvious that 1 unifies with $\mathbb{N}$, and 2 with $\mathbb{N} \to \mathbb{N}$, because $\mathbb{N} \to \mathbb{N} \to \mathbb{N} = \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, i.e. $\to$ is a right associative operator.

$$\begin{aligned} 1 &\longmapsto \mathbb{N} \\ 2 &\longmapsto \mathbb{N} \to \mathbb{N} \end{aligned}$$

Using this substitution, we can simplify some of the remaining equations

$$\begin{aligned} 2 \equiv 7 \to 8 &\longmapsto \mathbb{N} \to \mathbb{N} \equiv 7 \to 8 \\ 11 \equiv 1 \to 10 &\longmapsto 11 \equiv \mathbb{N} \to 10. \end{aligned}$$

From the first equation we get $7 \longmapsto \mathbb{N}$ and $8 \longmapsto \mathbb{N}$. And we can continue a similar process, until we find a solution or contradiction, like an equation $\mathbb{N} \equiv 1 \to 2$ which don't unify.

The unification algorithms mentioned above calculate a solution to this problem in programmable fashion. In the same way as *Gauss elimination* is an algorithm to calculate a solution to the system of linear equation, though a human could spot easier ways to solve the system.

In the $\lambda^{\to}$case, we would use the unification monad from the library directly as `ConstraintMonad` and solve the constraints on the go. Or even infer the types directly, as they can be decided with only local information, in $\lambda^{\to}$case. Yet, as we'll see later, the separation of concerns as we can use different techniques for solving different types of constraints. For example the syntactic structure of the term language could be different or/and we could use a different constraint solver.

Alternatively we can modify the unification procedure (or more general constraint solver) to dismiss unsatisfiable constraints, generating error on the go. In such a way we can continue the type-inference process, possibly reporting multiple errors. [35]

3.3 Hindley-Milner

Hindley-Milner (ML) [19, 37, 50], also known as Damas–Milner or Damas–Hindley–Milner, is a classical type system for the lambda calculus with parametric polymorphism. It's a subset language of the System F. Among HM's more notable properties is completeness and its ability to deduce the most general type of a given program without the need of any type annotations or other hints supplied by the programmer.

3.3.1 Declarative version

We begin by investigating a *declarative* version of Hindley-Milner type system. It's simple, elegant, and abstract. The syntax-directed variant in Section 3.3.3 is more bulky, but is more concrete and closed to an algorithm.

Definition 3.5 (HM syntax)**.** The syntax of the HM system is almost the same as in System F. The additional **let** case is added, to allow the inference of polymorphic bindings, called let-generalisation. The types are restricted to be only of the first rank, the universally quantified arguments aren't allowed. The whole syntax is presented in Figure 3.2. The function type is created by *type application*, when $D = \rightarrow$.

In the vanilla Hindley-Milner system, there aren't type-annotated lambda $\lambda(x : \sigma).\, e$. The term syntax is almost the same as in the untyped λ-calculus. Let-expressions are an additional redundant construct, which can be desugared into

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \quad \longmapsto \quad (x \to e_2)\ e_1$$

Yet, in the HM-system let-expressions have a special typing rule to allow type-inference of polymorphic let-bindings.

There are six rules in the declarative version of the Hindley-Milner system, they are presented in Figure 3.3. Note that types in the context could be

MONOTYPES	$\tau := \alpha$	*type variable*
	$\mid D\ \tau \ldots \tau$	*type application*
POLYTYPES	$\sigma := \tau$	*monotype*
	$\mid \forall \alpha.\, \sigma$	*universal quantification*
TERMS	$e := x$	*variable*
	$\mid e_1\ e_2$	*application*
	$\mid \lambda x.\, e$	*abstraction*
	$\mid$ **let** $x = e_1$ **in** e_2	*let expression*

Figure 3.2: Hindley-Milner syntax

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau}\ \textsc{Let}$$

$$\frac{\Gamma \vdash f : \tau \to \tau' \quad \Gamma \vdash x : \tau}{\Gamma \vdash f\ x : \tau'}\ \textsc{App} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e : \forall \alpha.\, \sigma}{\Gamma \vdash e : [\alpha / \tau]\, \sigma}\ \textsc{Inst} \qquad \frac{\Gamma \vdash e : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall \alpha.\, \sigma}\ \textsc{Gen}$$

Figure 3.3: Declarative Hindley-Milner type system

either mono- or polytypes. In particular we can abstract (ABS) only over monotypes, but in the LET rule we can introduce polytyped bindings

Rule INST quietly makes a very important point: the system is predicative, as type variables may range only over monotypes. We can see this from the fact that the type variables in INST are instantiated by τ types, not σ types.

Example 3.6 (Typing *id*)**.** In the Hindley-Milner system the $\forall$ quantifier is always the outer one. For example we cannot write a function which takes the generic *id* function as an argument, it has to be specialised, yet the type of the *id* function itself can be deduced.

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \in x : \alpha}{x : \alpha \vdash x : \alpha}\,\textsc{Var}}{\varepsilon \vdash \lambda x.\, x : \alpha \to \alpha}\,\textsc{Abs} \qquad \alpha \notin FV(\varepsilon)}{\varepsilon \vdash \lambda x.\, x : \forall \alpha.\, \alpha \to \alpha}\,\textsc{Gen} \qquad \dfrac{}{id : \ldots \vdash id : \ldots}\,\textsc{Var}}{\varepsilon \vdash \textbf{let } id = \lambda x.\, x \textbf{ in } id : \forall \alpha.\, \alpha \to \alpha}\,\textsc{Let}$$

Given the $\vdash id : \forall \alpha.\, \alpha \to \alpha$, the expression $id\ id$ is typable in Hindley-Milner system, yet there are also other deductions for it in the System F. If we write down type abstractions and applications explicitly, it will be clear. First recall the *id* in System F:

$$id = \Lambda \alpha.\, \lambda x : \alpha.\, x : \forall \alpha.\, \alpha \to \alpha$$

Using it we can add type abstractions and applications at least in two ways for the $id\ id$ expression

$$id\ id \approx \Lambda \beta.\, id\ (\beta \to \beta)\ (id\ \beta) : \forall \beta.\, \beta \to \beta \tag{3.1}$$
$$id\ id \approx id\ (\forall \alpha.\, \alpha \to \alpha)\ id \tag{3.2}$$

The version in (3.1) is how we can type the expression in Hindley-Milner. However (3.2) variant isn't possible in Hindley-Milner, as there we can instantiate type-variables only with monotypes.

3.3.2 Subsumption and principal type

Odersky and Läufer use the term *subsumption* for "more polymorphic than" -relation [44]. For the next sections we'll need to define such relations properly.

Definition 3.7 ($\sigma \preceq \tau$ relation)**.** The $\preceq$ relation is between poly- and monotype.

$$\sigma \preceq \tau$$

means that the outer quantifiers of σ can be instantiated to give τ. Instantiation is used in the rule INST to instantiate a type of a polymorphic variable at its occurrence sites. This is a restricted version of $\preceq$ relation (Definition 3.11), where the right-hand-side is monotype.

Example 3.8. More concretely, taken the type of identity function, we can instantiate it to work on natural numbers:

$$\forall \alpha.\alpha \to \alpha \preceq \mathbb{N} \to \mathbb{N}$$

One might also need to compare two *polytypes*. For example:

$$\begin{aligned}
\mathbb{N} &\sqsubseteq \mathbb{N} \\
\mathbb{N} \to \mathit{Bool} &\sqsubseteq \mathbb{N} \to \mathit{Bool} \\
\forall.\, \alpha \to \alpha &\sqsubseteq \mathbb{N} \to \mathbb{N} \\
\forall.\, \alpha \to \alpha &\sqsubseteq \forall \beta.\, \mathit{List}\ \beta \to \mathit{List}\ \beta \\
\forall.\, \alpha \to \alpha &\sqsubseteq \forall \beta\ \gamma.\, (\beta, \gamma) \to (\beta, \gamma) \\
\forall \alpha\ \beta.\, (\alpha, \beta) \to (\alpha, \beta) &\sqsubseteq \forall \gamma.\, (\gamma, \gamma) \to (\gamma, \gamma)
\end{aligned}$$

The third example involves only simple instantiation, but the last three illustrate the general case. Notice that the number of quantified type variables in the left-hand type can be the same, or more, or fewer, than in the right-hand type, as the last three examples demonstrate.

Definition 3.9 ($\sqsubseteq$-relation)**.** Type-inference for arbitrary-rank types would need $\sqsubseteq$-relation. It can be defined using three rules [57]:

$$\frac{}{\tau \sqsubseteq \tau}\ \textsc{Mono} \qquad \frac{[\alpha/\tau]\sigma \sqsubseteq \sigma'}{\forall \alpha.\, \sigma \sqsubseteq \sigma'}\ \textsc{Spec} \qquad \frac{\sigma \sqsubseteq \sigma' \qquad \alpha \notin FV(\sigma)}{\sigma \sqsubseteq \forall \alpha.\, \sigma'}\ \textsc{Skol}$$

Example 3.10. To illustrate the relation, let us see the proof of

$$\forall.\, \alpha \to \alpha \sqsubseteq \forall \beta\ \gamma.(\beta, \gamma) \to (\beta, \gamma)$$

First we'll use SKOL to skolemise β and γ, checking that they are not free in $\forall \alpha.\, \alpha \to \alpha$, and then use SPEC to instantiate α with (β, γ).

$$\cfrac{\cfrac{\cfrac{\cfrac{}{(\beta,\gamma) \to (\beta,\gamma) \sqsubseteq (\beta,\gamma) \to (\beta,\gamma)}\ \textsc{Mono}}{\forall \alpha.\, \alpha \to \alpha \sqsubseteq (\beta,\gamma) \to (\beta,\gamma)}\ \textsc{Spec}, \alpha \mapsto (\beta,\gamma)}{\forall \alpha.\, \alpha \to \alpha \sqsubseteq \forall \gamma.\, (\beta,\gamma) \to (\beta,\gamma)}\ \textsc{Skol}}{\forall \alpha.\, \alpha \to \alpha \sqsubseteq \forall \beta\ \gamma.\, (\beta,\gamma) \to (\beta,\gamma)}\ \textsc{Skol}$$

However, we rely only on a simpler $\leq$ -relation in this work. It will be useful in Section 3.4.

Definition 3.11 ($\leq$-relation)**.** $\sigma \leq \sigma'$ relation means that the *some* of outer quantifiers of σ can be instantiated to give σ'. It's similar to $\sqsubseteq$-relation, but without SKOL rule:

$$\frac{}{\tau \leq \tau}\ \textsc{Mono} \qquad \frac{[\alpha/\tau]\sigma \leq \sigma'}{\forall\alpha.\,\sigma \leq \sigma'}\ \textsc{Spec}$$

The relation symbols act as mnemonic for how "big" the relations are:

$$\sigma \preceq \tau \Rightarrow \sigma \leq \tau$$
$$\sigma \leq \sigma' \Rightarrow \sigma \sqsubseteq \sigma'$$

Definition 3.12 (Principal type)**.** The Hindley-Milner rules allow one to deduce different types for one and the same expression. For example

$$\Gamma \vdash \lambda x \to x : \forall\alpha.\alpha \to \alpha$$
$$\Gamma \vdash \lambda x \to x : \mathbb{N} \to \mathbb{N}.$$

The Hindley-Milner system has the *principal types* property. In other words, for all terms typable in a particular context, there is some "best" type for that term.

If some σ' exists such that $\Gamma \vdash x : \sigma'$, then there is σ (the principal type of x in context Γ such that

$$\Gamma \vdash x : \sigma$$
$$\forall\sigma'', \Gamma \vdash t : \sigma'' \Rightarrow \sigma \sqsubseteq \sigma''$$

The principal types are very import in practice. It means that an implementation can infer a single, principal type for each `let`-bound variable, that will work regardless of the contexts in which the variable is subsequently used.

3.3.3 Syntax directed version

Each rule in Figure 3.3 has a distinct syntactic form in its conclusion, except for two: GEN (generalisation) and INST (instantiation). Because these two have the same syntactic form in their premise as in their conclusion, one can

apply them pretty much anywhere; for example, one could alternate GEN and INST indefinitely. This flexibility makes it hard to turn the rules into a type-inference algorithm. For example, given a term, say $\lambda x.\, x$, it is not clear which rules to use, in which order, to derive a judgment $\vdash \lambda x.\, x : \sigma$ for some σ.

If all the rules had a distinct syntactic form in their conclusions, the rules would be in the so-called syntax-directed form, and that would, in turn, fully determine the shape of the derivation tree for any particular term t. This is a very desirable state of affairs, because it means that the steps of a type inference algorithm can be driven by the syntax of the term, rather than having to search for a valid typing derivation.

A common treatment of HM uses such a syntax-directed rule system due to Clement [15]. In this system, we merge INST into VAR rule and GEN rule into LET rule. The resulting rules are shown in Figure 3.4. As the rules are merged, a variable lookup always produce an instantiated monotype, and let-bindings will always be generalised. The let-generalisation is also determined to always produce the most general type by quantifying over all monotype variables in τ, that are not bound in Γ.

$$\frac{x : \sigma \in \Gamma \qquad \sigma \preceq \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash x : \tau}{\Gamma \vdash f\ x : \tau'}\ \textsc{App} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma, x : \forall \bar{\alpha}.\, \tau \vdash e_1 : \tau' \qquad \bar{\alpha} = FV(\tau) - FV(\Gamma)}{\Gamma \vdash \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau'}\ \textsc{Let}$$

Figure 3.4: Syntax-directed Hindley-Milner type-system

Example 3.13 (Typing *id* directed by syntax)**.** Recall Example 3.6, there the LET rule is followed by the GEN rule. In a syntax directed system these two derivations will merge into one.

$$\dfrac{\dfrac{\dfrac{x : \alpha \in x : \alpha \qquad \alpha \preceq \alpha}{x : \alpha \vdash x : \alpha}\ \textsc{Var}}{\varepsilon \vdash \lambda x.\, x : \alpha \to \alpha}\ \textsc{Abs} \qquad \dfrac{id \in \ldots \qquad \forall \alpha.\, \alpha \to \alpha \preceq \beta \to \beta}{id : \forall \alpha.\, \alpha \to \alpha \vdash id : \beta \to \beta}\ \textsc{Var}}{\varepsilon \vdash \textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id : \beta \to \beta}\ \textsc{Let}$$

The $\forall \alpha.\, \alpha \to \alpha \preceq \beta \to \beta$ is given by instantiating α with β.

As we can see, we cannot derive a polymorphic type for the expression in a syntax-directed system. The syntax-directed system is incomplete. However, a weaker version of completeness is provable

$$\Gamma \vdash^{declarative} e : \sigma \Rightarrow \Gamma \vdash^{syntax} e : \tau \ \wedge \sigma \preceq \tau$$

implying that one can derive the principal type for an expression in a syntax directed type-system, if one generalises the type in the end.

From the point-of-view of this thesis, the ML is a very nice example, where we sacrifice some of the expressiveness of the parent type-system (System F) as the trade-off for more powerful tooling. In this case complete type-inference!

The syntax-directed formulation is very close to actual *Algorithm W*, but we leave its presentation, as we will concentrate on constraint based inference in the next section.

3.3.4 Hindley-Milner as a Pure Type System

The Hindley-Milner type-system can be expressed as a Pure Type System. The key here is to realise that there are two kinds (or sorts) of types: monotypes, M and polytypes P, instead of just one $\star$. We can abstract over monotypes, generating polytypes.

We don't need $\Box$ as we don't abstract over polytypes, especially we cannot have $P : \Box$.

Definition 3.14 (Hindley-Milner as a Pure Type System)**.**

$$\begin{array}{ll} \mathcal{S} & \mathsf{M}, \mathsf{P} \\ \mathcal{A} & \mathsf{M} : \mathsf{P} \\ \mathcal{R} & (\mathsf{M}, \mathsf{M}, \mathsf{M}), (\mathsf{P}, \mathsf{M}, \mathsf{P}), (\mathsf{P}, \mathsf{P}, \mathsf{P}) \end{array}$$

Let's see what $(\mathsf{P}, \mathsf{M}, \mathsf{P})$ means in terms of PRODABS (Remark 2.22):

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{P} \qquad \Gamma, x : \mathsf{M} \vdash e : \beta : \mathsf{M}}{\Gamma \vdash (\lambda x : \mathsf{M}.\, e) : (\Pi x : \mathsf{M}.\, \beta) : \mathsf{P}}$$

If we abstract over a monomorphic type variable x, we get an expression of type $\Pi x : \mathsf{M}.\, \beta = \forall x.\, \beta$, which is a polymorphic type. The $(\mathsf{P}, \mathsf{P}, \mathsf{P})$ rule is similar, but the abstracted type is already polymorphic. The remaining $(\mathsf{M}, \mathsf{M}, \mathsf{M})$ is used to form ordinary functions.

Remark 3.15 (Hindley-Milner is a subsystem of ITT)**.** Take $\mathsf{M} = \mathcal{U}_0$ and $\mathsf{P} = \mathcal{U}_1$, i.e. the first and second universes of the ITT universe tower (2.4.4).

Remark 3.16 (Hindley-Milner is a subsystem of System F)**.** This isn't as simple as with ITT, as we cannot directly smash monomorphic and polymorphic types into just "types", i.e. $\mathsf{M} = \mathsf{P} = \star$, as we'd get a $\star : \star$ axiom. We'd need to make a more complicated simultaneous rewrite

$$\begin{aligned} \mathsf{M} &\mapsto \star \\ \mathsf{P} &\mapsto \star \\ \mathsf{M} : \mathsf{P} &\mapsto \star : \Box \\ (\mathsf{P}, \mathsf{M}, \mathsf{P}) &\mapsto (\Box, \star, \star) \\ (\mathsf{P}, \mathsf{P}, \mathsf{P}) &\mapsto (\Box, \star, \star). \end{aligned}$$

In other words, we'll rewrite some P to $\star$, and others to $\Box$. We'll omit the proof that this works, but it's a straight-forward inductive proof.

Remark 3.15 and Remark 3.16 highlight an interesting fact: Hindley-Milner is a subsystem of two other type-systems which aren't comparable to each other.

3.4 Constraint based inference for higher order calculi

The Helium Haskell compiler work [35, 36] have shown that the constraint based approach is viable for polymorphic languages, and can even be scripted to achieve better type errors. We take this approach even further where the type-system itself is modified to work better!

As in $\lambda^{\to}$ inference, we'll develop a constraint language from the typing rules. We will use the syntax directed version (Section 3.3.3). As there are additional judgments, we'll need to introduce new types of constraints:

$$\tau_1 \equiv \tau_2 \mid \sigma \preceq \tau \mid \tau \lhd_\Gamma \sigma$$

An equality constraint ($\tau_1 \equiv \tau_2$) is already familiar. The other two kinds of constraints are used to deal with the polymorphism. An explicit instance constraint $\sigma \preceq \tau$ states that τ has to be an instance of σ. This constraint is related to rule VAR. An implicit instance constraint $\tau \lhd_\Gamma \sigma$, which expresses that σ should be a generalisation of the monomorphic type τ

with respect to the set of type variables in Γ, i.e. quantifying over the other type variables, $FV(\tau) - FV(\Gamma)$. This constraint is used to deal with LET rule. This constraint language is a bit different to the one used in the Helium compiler [35], in particular our $\lhd_\Gamma$ constraint. The constraint generation is shown in Figure 3.5. Constraints generated for ABS and APP rule are the same as in $\lambda^{\to}$ case (Figure 3.1), the VAR and LET use new, just introduced constraints.

$$\frac{x : \sigma \in \Gamma \qquad \sigma \preceq \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \begin{aligned} [\![x]\!]_{x:\sigma} &\equiv \rho \\ \sigma &\preceq \rho \end{aligned}$$

$$\frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash x : \tau}{\Gamma \vdash f\ x : \tau'}\ \textsc{App} \qquad \begin{aligned} [\![f\ x]\!]_\Gamma &\equiv \rho \\ [\![f]\!]_\Gamma &\equiv [\![x]\!]_\Gamma \to \rho \end{aligned}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'}\ \textsc{Abs} \qquad \begin{aligned} [\![\lambda x.\, e]\!]_\Gamma &\equiv \rho \\ \rho &\equiv \rho' \to [\![e]\!]_{\Gamma, x:\rho'} \end{aligned}$$

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma, x : \forall \bar{\alpha}.\, \tau \vdash e_1 : \tau'}{\Gamma \vdash \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau'}\ \textsc{Let} \qquad \begin{aligned} [\![\textbf{let}\ x = e_0\ \textbf{in}\ e_1]\!]_\Gamma &\equiv \rho \\ \rho &\equiv [\![e_1]\!]_{\Gamma, x:\rho'} \\ [\![e_0]\!]_\Gamma &\lhd_\Gamma \rho' \end{aligned}$$

Figure 3.5: Constraint generation for HM type system

Example 3.17 (Typing *id* using constraint based inference)**.** Let's once again infer a type of *id* function. First we gather all the constraints:

$$\begin{aligned} [\![\textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id]\!] &\equiv 1 \qquad & 1 &\equiv [\![id]\!]_{id:2} \qquad & [\![\lambda x.\, x]\!] &\lhd_\varepsilon 2 \\ [\![\lambda x.\, x]\!] &\equiv 3 \qquad & 3 &\equiv 4 \to [\![x]\!]_{x:4} \\ [\![x]\!]_{x:4} &\equiv 5 \qquad & 4 &\preceq 5 \\ [\![id]\!]_{id:2} &\equiv 6 \qquad & 2 &\preceq 6 \end{aligned}$$

Here we know that variable 4 is monomorphic, so $4 \preceq 5$ constraint is equivalent with $5 \equiv 4$. After substituting all type equivalences we get

$$\begin{aligned} [\![\textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id]\!] &\equiv 6 \\ 4 \to 4 &\lhd_\varepsilon 2 \\ 2 &\preceq 6 \end{aligned}$$

Now all variables on the left side of $4 \to 4 \lhd_\varepsilon 2$ are almost "solved", i.e. are present only in that single equation, we can simplify $\lhd_\varepsilon$ constraint into $2 \equiv \forall\alpha.\, \alpha \to \alpha$. After that, we can instantiate 6 with fresh variables, completing the inference process with result $[\![\textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id]\!] \equiv \beta \to \beta$.

For our further developments, we won't use the full power of Hindley-Milner inference. We will omit the let-generalisation, as it will simplify the constraint language. Especially $\lhd_\Gamma$ constraint requires special treatment in the constraint solver [35]. By type-inferencing bindings one-by-one, requiring not-yet introduced functions to be given types explicitly (mutual recursion), and generalising closed types, we will need only a slightly more type-annotations [70]. Also in higher-order calculi subsumption becomes even more tricky.

- If the type signature is given: **let** $x : \sigma = e_0$ **in** e_1, we generate constraints $\rho = [\![e_1]\!]_{\Gamma,x:\sigma}$ and $[\![e_0]\!] \equiv \sigma$:

$$\begin{array}{r} [\![\textbf{let } x = e_0 \textbf{ in } e_1]\!]_\Gamma \equiv \rho \\ \rho \equiv [\![e_1]\!]_{\Gamma,x:\rho'} \\ [\![e_0]\!]_\Gamma \not\lhd_\Gamma \rho' \\ \equiv \sigma \end{array} \Longrightarrow \begin{array}{r} [\![\textbf{let } x = e_0 \textbf{ in } e_1]\!]_\Gamma \equiv \rho \\ \rho \equiv [\![e_1]\!]_{\Gamma,x:\sigma} \\ [\![e_0]\!]_\Gamma \equiv \sigma \end{array}$$

- if we are on the top level, i.e. context is empty, we infer a type of $[\![e_0]\!] = \rho'$, generalise all variables, and continue with type-inference of $e_1\ [\![x]\!] \lhd_\epsilon [\![e]\!]$. We can further relax this definition by treating all closed terms as "top-level" expressions.
- otherwise we use equality constraints, as if **let** would be desugared into $(\lambda x.\, e_1)\ e_0$.

Using this approach we can still infer types of *id* or *id id*. Actually coming up with a simple and not contrived example where we must give type signature to local let binding is not trivial.

Example 3.18 (Let should not be generalised)**.** We can start with slightly modified example from *Let should not be generalised* [70].

$$\begin{array}{l} \mathit{wuggle}\ x = \textbf{let } \mathit{singleton}\ y = [y] \\ \qquad\qquad \textbf{in } (\mathit{singleton}\ x, \mathit{singleton}\ \texttt{'w'}) \end{array}$$

If we don't generalise types of the closed expression, our approach will infer the type $\mathit{Char} \to ([\mathit{Char}], [\mathit{Char}])$. However *singleton* definition is a closed expression, so we can generalise the type, inferring the type $\mathit{wuggle} : \forall t.t \to ([t], [\mathit{Char}])$.

We need a definition with local, non-closed binding. Such situations occur for example when we close over some configuration object:

$$
\begin{array}{l}
\mathit{pipeline}\ n\ (xs, ys) = \\
\quad \textbf{let}\ \mathit{process} = \mathit{drop}\ n \\
\quad \textbf{in}\ (\mathit{process}\ xs, \mathit{process}\ ys)
\end{array}
$$

GHC infers a type $\mathit{pipeline} : \forall a\ b.\mathit{Int} \to ([a], [b]) \to ([a], [b])$, and the type of *process* is obviously $\mathit{process} : \forall c.[c] \to [c]$. However with `MonoLocalBinds` extension[1] on, which makes GHC infer less polymorphic types for local bindings by default, the inferred type is less general $\mathit{pipeline} : \forall a.\mathit{Int} \to ([a], [a]) \to ([a], [a])$, because the *process* function doesn't have a closed type anymore $\mathit{process} : [a] \to [a]$, forcing *xs* and *ys* to have the same type. However if we give a type signature to *pipeline*, we easily spot the invalid inference result, which is also easy to fix:

$$
\begin{array}{l}
\mathit{pipeline} :: \forall a\ b.\mathit{Int} \to ([a], [b]) \to ([a], [b]) \\
\mathit{pipeline}\ n\ (xs, ys) = \\
\quad \textbf{let}\ \mathit{process} :: \forall c.[c] \to [c] \\
\qquad \mathit{process} = \mathit{drop}\ n \\
\quad \textbf{in}\ (\mathit{process}\ xs, \mathit{process}\ ys)
\end{array}
$$

In our opinion, when working in a language with a powerful (and complex) type-system, one should always specify the types of the top-level definitions. Even if, it's well specified, how and what types inference engine should infer, the behaviour might has surprising corner cases. Also the implementations of the algorithms aren't always bug free.

We can adopt the HM approach to work with higher order calculi, like System F, by switching $\preceq$ in VAR to $\leq$ or even $\sqsubseteq$, this will give us ability to pass polymorphic values as arguments.

3.5 Conclusion

In this chapter we have gone through basic ideas behind type-inference. We introduced a constraint based type-inference approach, which is flexible enough to be modifiable for different type systems. In the next chapters we will use it to infer types in new domain specific languages.

[1] Which is implied by commonly used `GADTs` and `TypeFamilies` extensions.

Chapter 4

TypedLab

In this chapter, we describe the two variants TYPEDLAB language, suitable as an intermediate language for a MATLAB linter or compiler. The language presented here is based on $\lambda^{\rightarrow}$ and developed using methods from previous chapters. Also we'll see how TYPEDLAB could be extended further.

The reason to use intermediate language here, is to ensure correctness: if we can elaborate surface language into correct intermediate language code, the original program is correct to some degree. Also if we later implement a compiler, intermediate language would simplify that task as well.

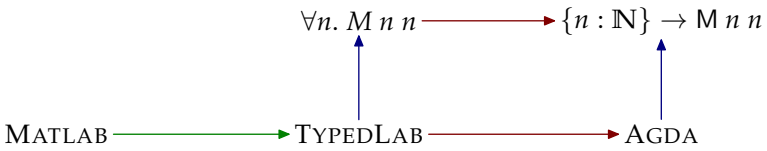

Figure 4.1: Matlab to Typedlab to Agda

One such pipeline is shown in Figure 4.1. We *could* compile TYPEDLAB further into AGDA. As we can describe TYPEDLAB as a closed universe inside AGDA, we could prove correctness of program transformations. Alternatively, we could compile MATLAB code to high-performance C++. These additions are out of the scope of this thesis.

4.1 Matlab

MATLAB is a multi-paradigm numerical computing environment. The same name is used for an actual programming language used in MATLAB environment. Considering its origin [51] it's natural that MATLAB is a dynamically typed language. Retrofitting the type system for the whole language is an enormous task. However, for many MATLAB programs, with mostly linear algebra computations, the needed system is simple.

Example 4.1. MATLAB is a simple imperative language.

```
rhs = boundaryCondition;
D = problemOperator;
u = D\rhs;
```

In the listing above, we assign `boundaryCondition` to variable `rhs`, and `problemOperator` to `D`. On the last line we use backslash operator (`mldivide`) to solve the system of linear equations `D * u = rhs`.

Here we can infer that `D` is a $n \times m$ matrix, and `rhs` is a n-vector. However, there are other, less strict type assignments for this example.

It would be beneficial to have vectors and matrix dimensions in the type-system, so the type-checker (or dimension-checker in this case) could prevent common *mismatching dimensions* errors. We are unaware of any widely used type checking based tools developed particularly for Matlab or Octave[1]. There are libraries in other languages [2], which let one specify the dimensions on the type level, yet all of them require explicit type annotations and/or are somehow limited.

In the rest of the chapter, we will show how to write a simple type-checker for a subset of MATLAB language, increasing the language step-by-step.

[1] GNU Octave is a high-level interpreted language, primarily intended for numerical computations. The Octave language is quite similar to Matlab so that most programs are easily portable.`https://www.gnu.org/software/octave/`

[2] C++: Eigen library has `Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>` class. HASKELL: `ghc-typelits-natnormalise` GHC type-checker plugins.

4.2 TypedLab-1: $\mathbb{N}$ indexes

We begin with the simplest system possible which allows us to encode dimensions in the type system.

INDEX	$n = x$	*number variable*
	$= 0, 1, \ldots$	*number constants*
TYPE	$\tau = \mathbb{M}\ n\ n'$	*rectangular matrices*
	$= \tau \to \tau$	*functions*
	$= \forall n.\, \tau$	*universal quantification over indexes*

We have type-level natural numbers to encode the dimensions, and "ordinary" types, which can be scalars, functions, and parametrised by natural number indexes. This system resembles Hindley-Milner in its structure, but is simpler as we quantify over type-variables of a totally different kind. This type-system is quite simple and we can encode it as a Pure Type System.

Definition 4.2 (Typedlab-1 as a Pure Type System)**.**

$$\begin{array}{ll} \mathcal{S} & \star, \Box, \mathbb{N} \\ \mathcal{A} & \star : \Box \\ \mathcal{R} & (\star, \star, \star), (\mathbb{N}, \star, \star) \end{array}$$

We also have other axioms: collection of natural number type-indexes, $0 : \mathbb{N}$, $1 : \mathbb{N}$ etc. Also we have $\mathbb{M} : \mathbb{N} \to \mathbb{N} \to \star$ type constructor representing square matrices.

The $(\mathbb{N}, \star, \star)$ PTS rule is quite similar to the rule $(\Box, \star, \star)$ in System F. The $\star$ and $\mathbb{N}$ are, however, unrelated: there isn't axiom relating them. The INST and GEN rule analogues are even simpler than in the Hindley-Milner system. We use N as λ analogue, to quantify over $\mathbb{N}$ indexes.

$$\frac{\Gamma \vdash \alpha : \forall n.\, \beta \qquad m : \mathbb{N}}{\Gamma \vdash \alpha\ m : [n/m]\ \beta}\ \textsc{Inst-}\mathbb{N} \qquad \frac{\Gamma \vdash x : \alpha \qquad n \notin FV(\Gamma)}{\Gamma \vdash \mathsf{N} n.\, x : \forall n.\, \alpha}\ \textsc{Gen-}\mathbb{N}$$

This type-system is as subsystem of ITT, and easily embeddable into AGDA, so we can conclude that this type-system is consistent. The embedding to AGDA in Appendix B.3.

As previously, we can traverse the syntax tree of the program gathering constraints. As we take a relaxed, non-let-generalising approach, we'll need

only an analogue of $\leq$ constraint, or create instantiation variables already during traversal. We'll need variables of two different kinds: $\mathbb{N}$ and $\star$ for dimensions and type-of-terms respectively. Unfortunately `unification-fd` library doesn't support variables of different kinds, so we need to do the sanity check, after the unification is done.

Remark 4.3 (Higher-order functions)**.** The TYPEDLAB definition allows us having functions of higher orders, like

$$\forall n.\, (\forall m.\, \mathbb{M}\ 1\ 1 \to \mathbb{M}\ m\ m) \to \mathbb{M}\ n\ n$$

Our constraint approach cannot infer definitions of such functions, and if we don't use $\leq$ constraint, it would be impossible to use them. Yet as TYPEDLAB is only a proof-of-concept, we didn't work on higher-order support.

So far there isn't anything complicated. This type-system resembles Hindley-Milner, but is even simpler.

4.3 TypedLab-2: $\mathbb{N}$ constraints

An attentive reader may notice that some of MATLAB-operators are more flexible, for example the already mentioned backslash operator. If the matrix on the left is not a square matrix then the equation is solved in the least squares sense. There are a few options: we can allow any matrices, making it impossible to catch any dimension bug. Or we can only allow matrices with more rows than columns, so the least squares solution would be unique. We can be even more precise by using a special name for the least squares solution. Anyway we'd need to be able to compare dimension variables. In Haskell syntax we'd like *leastsq* to have type

$$\mathit{leastsq} : \forall m\, n : \mathbb{N}.\, m \leq n \Rightarrow \mathbb{M}\ m\ n \to \mathbb{M}\ m\ 1 \to \mathbb{M}\ n\ 1$$

To allow this, we need to make a slight change to the language, introduce $\leq$-constraints, and allow having them in polymorphic types.

INDEX	$n = x$	*number variable*
	$= 0, 1, \ldots$	*number constants*
CONSTRAINT	$\mathcal{C} = n \leq n'$	*less-than constraints*
TYPE	$\tau = \mathbb{M}\ n\ n'$	*rectangular matrices*
	$= \tau \to \tau$	*functions*
	$= \forall n.\, \tau$	*universal quantification over indexes*
	$= \mathcal{C} \Rightarrow \tau$	*constraining indexes*

Definition 4.4 (Typedlab-2 as a Pure Type System)**.** We extend Definition 4.2 by adding constraint sort $\mathcal{C}$

$$\begin{array}{ll} \mathcal{S} & \star, \Box, \mathbb{N}, \mathcal{C} \\ \mathcal{A} & \star : \Box \\ \mathcal{R} & (\star,\star,\star), (\mathbb{N},\star,\star), (\mathcal{C},\star,\star) \end{array}$$

Also we have $\leq : \mathbb{N} \to \mathbb{N} \to \mathcal{C}$ constraint-constructor, we could have it as an axiom or in initial environment.

This system is also a subsystem of ITT, the embedding into AGDA is in Appendix B.4. The most interesting part is fromConstraint which embeds constraints into the AGDA type system.

The INST/GEN rules generated from $(\mathcal{C},\star,\star)$ rule are interesting:

$$\frac{\Gamma \vdash x : n \leq m \Rightarrow \beta \qquad \mathit{witness} : n \leq m}{\Gamma \vdash x\ \mathit{witness} : [n \leq m / \mathit{witness}]\ \beta}\ \textsc{Inst-}\mathcal{C}$$

$$\frac{\Gamma \vdash n \leq m : \mathcal{C} \qquad \Gamma, n \leq m : \mathcal{C} \vdash e : \beta : \star}{\Gamma \vdash (\mathsf{C}\, \mathit{witness} : n \leq m.\, e) : (n \leq m \Rightarrow \beta) : \star}\ \textsc{Gen-}\mathcal{C}$$

where *witness* is a proof term that witness that the constraint is satisfiable. The C is λ and N analogue, we use different symbol to abstract over different kinds of terms. We won't generate the actual proof-values as they are irrelevant, we are only interested whether *any* proof exists, i.e. whether $n \leq m$ type is inhabited.

Remark 4.5. We don't have conjuctions in the constraint language, but we can still accumulate many constraints by chaining them. $n \leq m \Rightarrow n' \leq m' \Rightarrow \alpha$ is equivalent to $n \leq m \wedge n' \leq m' \Rightarrow \alpha$.

The constraint language at this point is still simple. We must add $\mathcal{C}$ also to the constraint language. We can always simplify constraints by reducing cycles into equality constraints:

$$n_1 \leq n_2 \leq \cdots \leq n_m \leq n_1 \Rightarrow n_1 = n_2 = \cdots = n_m$$

If we add also $<$ constraint, then cycles envolving at least one $<$ would lead to the contradiction: $a < b \leq a \Rightarrow \bot$. However, it's simpler to use a ready-made decision procedure for Presburger arithmetic (4.4), so we can support a much richer constraint language out-of-the-box.

Some decision procedures for Presburger arithmetic support problem simplification [62]. However we discuss what to do, if simplificiation isn't built-in. Or we don't even have the first-order decision procedure. There are various different approaches to how to deal with newly introduced constraints:

- The simplest approach would be to accumulate the constraints over the variables, checking them only when they are fully instantiated. That would lead to many overlapping constraints, like $1 \leq n \wedge 2 \leq n$ and constraint sets that cannot be satisfied, like $2 \leq m \wedge m \leq 1$. Also error messages will occur much later, far from the error source.

- An opposite approach is to check decidable constraints right away.For example constraints in the Presburger arithmetic. If they are provable, we can dismiss them. If they are contradictionary, we report an error. Otherwise we leave them.

- If we have a list of conjuctive constraints, we can try to eliminate single elements, thus simplifying the overall constraint, by trying whether the rest of the constraints implies the first one. $2 \leq n \Rightarrow 1 \leq n$, so $1 \leq n \wedge 2 \leq n \equiv 2 \leq n$.

- Ask a user (programmer) to supply a list of constraints, which are "known" to be true. For example $\forall n : \mathbb{N}.\, n \leq n \times n$ cannot be decided automatically, but the special proof has to be given. As we don't need explicit proof terms, only stating additional "axioms" is enough. Obviously the user of the system might easily break it, by introducing e.g. $1 \equiv 0$ constraint, but that's the user's fault then.

In this section we added constraints on the indexes to the TYPEDLAB. In the next section, we'll introduce *Presburger Arithmetic*, a decidable theory that could be used for the integral constraints. In Section 4.5 we'll discuss how the language could be further expanded, if needed.

4.4 Presburger arithmetic

Presburger arithmetic is the first-order theory of the natural numbers containing addition but no multiplication. It is therefore not as powerful as Peano arithmetic. However, it is interesting because unlike the case of Peano arithmetic, there is an algorithm that can decide if any given statement in Presburger arithmetic is true [16, 61, 62].

Presburger arighmetic consists of the following axioms:

$$\neg(0 = x + 1)$$
$$x + 1 = y + 1 \to x = y$$
$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$
$$P(0) \wedge (\forall x, P(x) \to P(x + 1)) \to \forall y, P(y).$$

Alternatively, we can say that we can decide P (give a proof term for $P \vee \neg P$) for all P made from the following syntax:

variables: $n, m, \ldots = 0 \mid 1 \mid n + m$
formulas: $P = n \leq m \mid \forall n.P \mid \exists n.P \mid P \wedge Q \mid P \vee Q \mid \neg P.$

Example 4.6 (Less-than relation)**.** We can encode less-than relation by using an existential quantification:

$$x \leq y \to \exists z, x + z = y, \quad x, y, z \in \mathbb{N}$$

As with the unification, how exactly the satisfiability problem is decided, is not important, it's enough to know that there are algorithms for that [8, 16], which are packages into the libraries[3].

Alternatively we can plug-in an SMT-solver. All popular solvers can decide satisfiability of Presburger arithmetic formulas. Incorporating an SMT-solver gives us the ability to reason about additional theories almost for free, this might be useful too.

4.5 TypedLab-3: further extensions

The TYPEDLAB-2 is already a quite powerful system in its domain. There aren't obvious additional features needed to work with linear algebra.

[3]For example: `presburger-package`

Yet, we discuss a few extensions which would be easy to add into the TYPEDLAB-framework.

The first direction is to explore the higher-order functions mentioned in Remark 4.3. We don't know how common these are in a software written in MATLAB. Another direction is to extend base calculus into Hindley-Milner. This can be done in a straigth-forward way, by having four sorts: $\mathcal{S} = \mathsf{M}, \mathsf{P}, \mathbb{N}, \mathcal{C}$

Integer ring. Some arithmetic with multiplication is also decidable. [32], for example we can prove identities like $(x+y)^2 = x^2 + 2xy + y^2$, the latter is *sum-of-products* form of the former. The sum-of-products form is a normal form, so we can easily compare expressions for equivalence using structural equality.

Units of measure. In physical applications, adding units to the matrices might be sensible, and avoid silly "forgotten unit-conversion" errors. There are works on decidability of unit calculus [33, 34, 43]. In our framework, we'd add another sort, $\mathcal{Y}$, which would contain units, and change the matrix type to be parametrised on it as well: $\mathbb{M} : \mathbb{N} \to \mathbb{N} \to \mathcal{Y} \to \star$. Also we'd need to be able to abstract over units, even to define the addition operation. The frontend of TYPEDLAB would need only slight modifications, also the unit-equality constraints could be solved quite independently.

Regular expressions. Regular expressions is an algebraic description of the regular languages. Regular languages have nice properties: they are closed under concatenation, union, complement, and intersection. [38] It's possible to compare their descriptions for the equivalence, whether two regular expressions denote the same language, e.g. $a^*a^* \equiv_? a^*$. The text-book solution is to generate minimal deterministic finite automata (DFA) of both expressions, and compare them for equality. There also other approaches [1, 2], the one based on partial derivates has an extremely simple idea:

- Testing the equivalence of two regular expressions, is the same as testing that both are greater-or-equal than the other: $a \equiv b \Leftrightarrow a \subseteq b \wedge b \subseteq a$.
- Testing of less-than relation can be done with help of the intersection operation: $a \subseteq b \Leftrightarrow a - b \equiv \emptyset \Leftrightarrow a \wedge \neg b \equiv \emptyset$. Then the equivalence relation test becomes a comparison with an empty languge: $a \equiv b \Leftrightarrow$

$(a \wedge \neg b) \vee (\neg a \wedge b) \equiv \emptyset$[4].

- Without intersection or negation operators the emptyness test of regular-expressions is a trivial structual check, as every other operator has an empty regex as either an annihilating or identity element:

$$\emptyset \cdot a \equiv a \cdot \emptyset \equiv \emptyset$$
$$\emptyset \vee a \equiv a \vee \emptyset \equiv a$$
$$\emptyset^* \equiv \emptyset.$$

  A negation operator however requires us to decide if a accepts everything: $a \equiv \Sigma^*$. Formally: $\neg a \equiv \emptyset \Leftrightarrow a \equiv \Sigma^*$. That's a difficult property to decide, consider for example a single character alphabet $\Sigma = \{a\}$ and a regular expression $(aa)^* \vee a(aa)^*$. Equally difficult is to decide, if the intersection is empty: $a \wedge b \equiv \emptyset$.

- However we can take regular expression derivates, derivates of derivates, etc. forming a finite set of "states" reachable from the initial state. The resulting regular expression derivatives are "the rest of the string accepted from this state". Brzozowski [10, 56] proved that this procedure terminates. To decide if the regular expression denotes empty language $a \equiv \emptyset$, it's enough to check that every state isn't terminal, it doesn't accept an empty string i.e. isn't *nullary*.

The ordinary strings tagged with regular expression, could be used to have statically checked enumerations in dynamic languages, where we use strings to denote different enumeration values. For example `eig` function in MATLAB, which returns a column vector containing the generalized eigenvalues of two square matrice. It may take an algorithm value as a third option:

$$eig : \forall n.\, \mathbb{M}\ n\ n \to \mathbb{M}\ n\ n \to \mathcal{S}\ \texttt{chol|qz} \to \mathbb{M}\ n\ 1$$

For example, if we use `eig` incorrectly:

```
e = eig(A, B, "cho");
```

the type checker will notice that `chol|qz` regular expression doesn't match `"cho"`, and will report a type error.

[4]The inversion of $\wedge$ to $\vee$ feels unnatural, but consider `a == false && b == false` $\Leftrightarrow$ `a || b == false`

Sets. Sets of values with decidable equality is a less expressive version of regular expressions, we omit kleene star and concatenation. Alternatively we can leave concatenation, as it's the kleene star operation which causes complexity when dealing with regular expressions. The sets of strings would be enough for an enumeration case mentioned above. However the kleene star might be useful when the enumeration is infinite, yet not $\Sigma^\star$, e.g. valid identifiers are often only a subset of all possible strings.

Sets could be further extended to *Finite maps* to support anonymous records:

$$\begin{aligned} &\textbf{type } \mathit{Person} = \{\texttt{name} : \mathit{String}, \texttt{phone} : \mathit{String}\} \\ &\mathit{printName} : \forall r.\, \{\texttt{name} : \mathit{String}\} \uplus r, \end{aligned}$$

where *printName* is *row polymorphic* function, which takes any record with field `name` of type *String* as an argument.

There are various design choices: whether fields can be masked, is there notion of missing field etc. We do not investigate row polymorphism further at this point, though it is required to complete the JSVERIFY story.

4.6 Conclusion

The TYPEDLAB is a proof-of-concept example, how one could design a type system to an isolated part of a bigger language. We have also seen various options how the type-system could be further expanded, still using the same tools we developed in the previous chapters.

We discovered recently related work which uses a similar approach by Wiik and Boström [73]. Wiik and Boström also use a Hindley-Milner inspired algorithm to infer types. They also note that dimensions and other types can be infered separately. Their work restricts the shape-polymorphism to the first-order, which is what inference can do. Also as they restrict themselves to the subset of MATLAB suitable for code generation. Higher-order functions are problematic in that context, because matrix types and shapes have to be determined statically. Their type-system also doesn't use $\leq$ constraints, but relies on predefined shape functions. In our approach, we have to use separate names for e.g. scalar-matrix multiplication, yet we can support *leastsq* type of functions.

In the next chapter we will go through development of type-system for JAVASCRIPT as it is used in the JSVERIFY library, used in real world applications. We'll discuss further type-system extension options and their implications.

Chapter 5

Typesson

TYPESSON is our attempt to add typing to the JAVASCRIPT language. There are other attempts to retrofit a type-system to JAVASCRIPT, for example *Dependent JavaScript* . DJS is surprisingly expressive, supporting various challenging features of the language. [13] However the JSVERIFY introduced in Section 5.2 uses only a small portion of language features is used, there are some conventions probably typical only to that library, which we'll discuss in Section 5.3. So in this chapter, and particularly in Section 5.4 we describe a relatively simple, yet expressive enough type-system which is suitable to type the JSVERIFY library.

5.1 JavaScript

JAVASCRIPT is a high-level, dynamic, untyped, and interpreted programming language. Alongside HTML and CSS, it is one of the three core technologies of World Wide Web content production; the majority of websites employ it and it is supported by all modern Web browsers without plug-ins. JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. [29] It has been standardized in the ECMAScript language specification. It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage, or graphics facilities, relying for these upon the host environment in which it is embedded. [24]

Some of the facilities of ECMAScript are similar to those used in other programming languages; in particular C, JAVA™, SELF, and SCHEME. [24]

The syntax of JAVASCRIPT is highly influenced by C and JAVA[1]:

```
function hello(name) {
    if (name! == null) {
        console.log("Hello, " + name);
    } else {
        console.log("Hello world");
    }
}
```

However, further in this chapter we concentrate on the semantically functional subset, i.e. the part of the language, we think is most influenced by SCHEME.

5.2 JSVerify

JSVERIFY is a property-based testing library, highly inspired by another library: QUICKCHECK [14]. While original QUICKCHECK is written in HASKELL, there are versions written in dynamic languages, for example in ERLANG [3, 63].

Back then, there wasn't a QUICKCHECK clone for JAVASCRIPT with enough features, most notably *shrinking*. After the library have found a counterexample, it would try to shrink it into hopefully a minimal example, which would be easier to understand. Randomly generated test-cases tend to have a lot of irrelevant details. As we leave generating test case to the machine, we also make it filter out the non-essentials of the failing cases.

Example 5.1 (Boolean function)**.** This example is taken from older revision of Software Foundations book [60]. We could prove the proposition by hand, as there are only four distinct $Bool \to Bool$ functions. Or we can let JSVERIFY generate inputs for us.

```
// forall (f: json -> bool, b: bool), f (f (f b)) ≡ f(b).
var boolFnAppliedThrice =
    jsc.forall("bool -> bool", "bool", function (f, b) {
        return f(f(f(b))) === f(b);
    });
```

[1] In the idiomatic JAVASCRIPT code there would be *name* = *name* || **null**; assignment as the first statement of the function , to verify the inputs. TYPESSON makes such checks unnecessary, especially for the internal functions.

```
jsc.assert(boolFnAppliedThrice);
// OK, passed 100 tests
```

In this case input types are finite, so we could evaluate all possible values exhaustively. However many practical cases are either infinite, or just too large, so we randomly generate *some* inputs, hoping that we'll find a counterexample if one exists.

At this point, let's introduce some types used in JSVERIFY.

- The result of the *jsc.forall* function is a *Property*, a thing that JSVERIFY tries to disproof, by randomly searching for a counter-example.
- Values of the type α for the tests are generated by a *Generator* α.
- If the counter-example is found, we try to make it smaller using *Shrink* α.
- And at the end we display the counterexamples using *Show* α.
- The can use these three parts together to build an *Arbitrary* α. Arbitraries are what the user of JSVERIFY works with most of the time. For example, they could be given to the *jsc.forall*, though in this example the small arbitrary specification language. to *jsc.forall*

In the rest of this chapter we will concentrate on how JSVERIFY is build internally, so we can "type-check" the implementation. It would be very unfortunate, if bugs in the code under the test aren't discovered because of the buggy test library.

5.3 JavaScript in JSVerify

The language used in JSVERIFY is a functional subset of JAVASCRIPT. While giving a type system for whole JAVASCRIPT is an enormous task

Example 5.1 illustrates the power of the dynamic typing, the simplified type of the *jsc.forall* function is

$$\forall \alpha\ \beta.\ [\mathit{ArbSpec}\ \alpha, \mathit{ArbSpec}\ \beta, [\alpha, \beta] \to \mathit{Bool}] \to \mathit{Property}$$

where α and β are instantiated with $(Bool) \to Bool$ and $Bool$ types respectively. The square brackets denote the function argument list. In TYPESSON we have multi-argument functions. An *ArbSpec* α is a string specification of an *Arbitrary* a. We can see many different language features used in JSVERIFY, further explained in this section: *nominal types* (Section 5.3.3) and *heterogeneous functions* (Section 5.3.2). One should however note, that this is a public API[2], this work is more interested in the internal uses, where different features are used like *type intersections* (Section 5.3.4) and "low-level" code (Section 5.3.5). Internals are also the part, where we can verify the typing, i.e. get the benefits of the static typing immediately.

5.3.1 Unused parts of the language

JAVASCRIPT is a pretty big language, and there are many features which we have found not useful or otherwise problematic for the JSVERIFY development. In this section, we'll list some of them.

EcmaScript 6. JavaScript was created at Netscape, which submitted the language for standardisation to the European Computer Manufacturer's Association. Because of trademark issues, the standardized version of the language was stuck with ECMASCRIPT name. Widely implemented version 3 of the ECMASCRIPT standard was published in 1999.. Version 5 was published in 2009, and version 6 in 2016.

JSVERIFY uses features from the fifth version. There are useful features in ECMASCRIPT6. For example *spread-operator* significantly reduces the cruft around multivariadic functions.

```
function sum(...args) {
    return [...args].reduce((x, y) ⇒ x + y, 0);
}

var arr = [1, 2, 3];
console.log(sum(1, 2, 3, ...arr)); // 12
```

Also it makes their usage syntactically obvious, in comparison to the manual *arguments* object slicing or *apply* method usage. We have to use ad-hoc support to "match" current use-cases in multivariadic functions.

Prototypical inheritance. JAVASCRIPT uses quite simple and powerful inheritance model. In the dynamic language we could have a *blessing*

[2]Application programming interface

function populating the object with all needed functionality:

```
function method1() {
    console.log(this.field);
}

function bless(obj) {
    obj.method1 = method1;
    return obj;
}

function mkObj(v) {
    return bless({ field : v });
}

var x = mkObj(42);
x.method1(); // 42
```

However there is a smarter way. If an accessed object member doesn't exist, lookup from the member from its *prototype*, following the chain until member is found or there aren't a prototype specified[3].

```
function Obj(v) {
    this.field = v;
}

Obj.prototype.method1 = function () {
    console.log(this.field);
}

var x = new Obj(42);
x.method1(); // 42
```

For example LUA language [41] uses the same approach, where prototypes are called *metatables*, as objects are "tables";

Yet in JSVERIFY we don't need inheritance, and we don't even exploit prototypes to provide common functionality for objects. There is a very simple reason for that. We need some of our objects to behave like functions, but we cannot construct function with other prototype than *Function*. ECMASCRIPT6 proxies could be used to achieve that functionality, but they cannot be transpiled to ECMASCRIPT5.

[3]It seems that prototype of *Function* is *Function* itself.

eval. The *eval* function is an ultimate dynamism provider. You can construct part of your program (as a string) and then *eval* it.

```
eval("console.log(42)"); // 42
```

Obviously such feature makes type-checking close to impossible, and there aren't justified use for *eval* in JSVERIFY.

Mutability. We highlighted some of JAVASCRIPT features, we don't use. At the end, we have to mention *mutability*. In JAVASCRIPT everything is mutable by default. In functional programming immutable structures are preferred, and there aren't much mutation going on in JSVERIFY. However from the point of view of type-checking mutability isn't a problem, so occasional use is easy to support, while totally avoiding it might turn out problematic.

5.3.2 Heterogeneous functions & containers

In Section 5.3.1 we briefly mentioned multiargument functions. In Haskell and many other statically typed languages we cannot have n-ary functions. We resolve the problem by introducing new names for variants of different arities:

$$\begin{aligned} &zipWith :: (a \to b \to c) \to [a] \to [b] \to [c] \\ &zipWith3 :: (a \to b \to c \to d) \to [a] \to [b] \to [c] \to [d] \end{aligned}$$

and

$$\begin{aligned} &liftA :: Applicative\ f \Rightarrow (a \to b) \to f\ a \to f\ b \\ &liftA2 :: Applicative\ f \Rightarrow (a \to b \to c) \to f\ a \to f\ b \to f\ c \\ &liftA3 :: Applicative\ f \Rightarrow (a \to b \to c \to d) \to f\ a \to f\ b \to f\ c \to f\ d \end{aligned}$$

In the JAVASCRIPT we only have a single *zipWith* function, using ECMASCRIPT6 pseudo syntax[4]:

```
function zipWith(...xs, f) {
    // implementation
}
```

It is possible (using type class trickery) to make variadic functions in HASKELL, but it's not elegant. But in JAVASCRIPT functions can take an

[4] in ECMASCRIPT6 spread argument must be the last formal parameter.

arbitrary amount of parameters, and the function application is always *saturated* i.e. all arguments are given. If we treat the argument list as a heterogenous list, we can write type for *zipWith* as:

$$zipWith : \forall \alpha s\ \beta.\ [\ldots Map\ List\ \alpha s, FoldFun\ \alpha s\ \beta] \to List\ \beta$$

The remaining question is: What are *Map* and *FoldFun*. The *Map f xs* could be written using *n-ary product*, *NP*. The name and ideas are influenced by `generics-sop` library [20]. Experience shows that *NP* is better suited than more direct *HList* formulation.

```
data NP (f :: ⋆ → ⋆) (xs :: [⋆]) where
  Nil  :: NP f '[]
  (:::) :: f x → NP f xs → NP f (x ': xs)
type family FoldFun (xs :: [⋆]) (b :: ⋆) where
  FoldFun '[]       b = b
  FoldFun (x ': xs) b = x → FoldFun xs b
type family Map (f :: ⋆ → ⋆) (xs :: [⋆]) where
  Map f '[]       = '[]
  Map f (x ': xs) = f x ': Map f xs
```

Using this definitions, it's quite simple to write (pseudo-)variadic *zipWith* in HASKELL, note how the inner function *zipWithN′* structure resembles a *foldl* definition:

```
zipWithN :: NP [ ] xs → FoldFun xs a → [a]
zipWithN xs f = zipWithN′ (repeat f) xs
  where
    zipWithN′ :: [FoldFun ys b] → NP [ ] ys → [b]
    zipWithN′ f Nil       = f
    zipWithN′ f (y ::: ys) = zipWithN′ (zipWith ($) f y) ys
{- Evaluates to [25, 37] -}
example :: [Int]
example = zipWithN
    ([1, 2, 3] ::: [4, 5] ::: [6, 7] ::: Nil)
    $ λa b c → a + b × c
```

It's possible to program using these definitions, but it's a bit inconvenient. The goal of TYPESSON, and domain specific type-systems in general, is to make the usage of advanced type level constructs simple.

More rigidly, $NP\ f : [\star] \to \star$ and $Map\ f : [\star] \to [\star]$ have different kinds. But we only use *Map*, so the notation is more lightweight. It's clear from the

context when we need to insert omitted *NP Id*[5]. It turns out that variadic functions and heterogenous lists are very common in JAVASCRIPT and in the JSVERIFY codebase in particularly. We add various type-level list constructs to TYPESSON, to deal with these idioms. The lack of spread-operator makes identifying some use-sites difficult, but not impossible.

5.3.3 Nominal types

Nominal typing means that two variables are type-compatible if and only if their declarations name the same type. In JAVASCRIPT and other dynamic languages, the typing is *structural*:

- **Nominal**: *Pet*
- **Structural**: {`name` : *String*, ...}

The TYPESSON has nominal types. There are only a handful of such types, and we list them all in the type-system definition. Nominal typing is useful at preventing accidental type equivalence, which allows better type-safety.

In addition, Nominal and explicitly available types allow us to add very specific typing rules, as we can treat types differently, even they have similar structure. As a consequence, we don't need to introduce row-types and row-polymorphism into our system, which keeps it conceptually simpler.

Nominal types together with intersections (Section 5.3.4) are used to implement ad-hoc polymorphism.

5.3.4 Intersections

Another property of dynamic languages is ad-hoc polymorphism using runtime information. For example we can have functions which uppercases the strings, and rounds up the numbers:

```
function adhoc(x) {
    if (typeof x === "string") {
        return x.toUpperCase();
    } else {
        return Math.ceil(x);
```

[5] *NP Id* (*Map f xs*) ≡ *NP f xs* as we will see later.

```
    }
}

console.log(adhoc("foo")); // FOO
console.log(adhoc(12.3)); // 13
```

In HASKELL we use type-class machinery to write ad-hoc polymorphic functions

```
class    Adhoc a      where adhoc :: a → a
instance Adhoc Text   where adhoc = Text.toUpper
instance Adhoc Double where adhoc = fromIntegral ∘ ceiling

ex1 :: Text
ex1 = adhoc "foo"  – "FOO"

ex2 :: Double
ex2 = adhoc 12.3   – 13
```

We could used something like type-classes to give a type to $ad - hoc$ in TYPESSON, However, we decided to use anonymous approach: intersection types. Instead of having class and instance definitions, we have a single intersection definition:

$$\begin{aligned} adhoc : [Number] &\to Number \\ \cap\, [String] &\to String \end{aligned}$$

This approach allows us to have type-signatures, which otherwise would require GHC extensions in HASKELL. Moreover, using unions it's easy to write a type for functions which behaves differently when called with different amount of the arguments.

Intersections introduce non-determinism into constraint solving process. At the moment we require that intersections are distinct, i.e. that only one can be satisfied. In the implementation, we use non-determinism monad as a base monad for the constraint solving, but there is also an additional reason: it's not always obvious from the syntax which rule to apply, as the TYPESSON system cannot be written in a syntax directed form.

Example 5.2 (Partially applicable functions)**.** Contrary to the heterogeneous functions in Section 5.3.2, we sometimes want functions to be partially applicable. This is convenient feature, as we can have more concise code,

as we don't need to write closures at the use-sites.

$$\begin{aligned} \mathit{shrinkPair} : \forall \alpha\ \beta.\, (\mathit{Shrink}\ \alpha, \mathit{Shrink}\ \beta) &\to \mathit{Shrink}\ (\mathit{Pair}\ \alpha\ \beta) \\ \cap (\mathit{Shrink}\ \alpha, \mathit{Shrink}\ \beta, \mathit{Pair}\ \alpha\ \beta) &\to \mathit{List}\ (\mathit{Pair}\ \alpha\ \beta) \end{aligned}$$

It can be used by passing directly three arguments, or only two and then third one to the returned function:

```
shrinkPair(shrinkBool, shrinkInt, [true, 1]); // [[false, 1], ...
shrinkPair(shrinkBool, shrinkInt)([true, 1]); // [[false, 1], ...
```

This kind of auto-currying is handy in practice, as often we construct *Shrink* and pass it forward, but sometimes we want to use it right after.

5.3.5 Low-level code

Some of the JSVERIFY code is quite "clever", taking the full advantage of the fact JAVASCRIPT is a dynamic language. For example, there are *curried2* and *curried3* utility functions, which has a complicated types:

$$\begin{aligned} \mathit{curried2} : \forall \alpha\ \alpha'\ \beta.\, [[\alpha, \alpha'] \to \beta, [\alpha, \alpha']] &\to \beta \\ \cap [[\alpha, \alpha'] \to \beta, [\alpha]] &\to [\alpha'] \to \beta \end{aligned}$$

The *curried2* function takes another function, f, and an argument list, and based on how many arguments there are, either fully or partially applies given function. The type is above is not a full truth, as there are ad-hoc rules to deal with nominal unary type constructors. The *curriedN*, used to define *curried2* and *curried3*, has very complicated definition

```
function curriedN(n) {
    var n1 = n - 1;
    return function curriedNInstance(result, args) {
        if (args.length === n) {
            return result(args[n1]);
        } else {
            return result;
        }
    };
}

var curried2 = curriedN(2);
var curried3 = curriedN(3);
```

and typing it this is non-trivial. These functions are used to define auto-curried functions, like *shrinkPair* from the previous section:

```
function shrinkPair(shrA, shrB) {
    var result = shrinkBless(function (pair) {
        // use shrA, shrB and pair
    });

    return curried3(result, arguments);
}
```

Giving *curriedN* a type in our framework isn't impossible. We'll need to make some extension particularly for this single use case though. More pragmatic approach is to give *curriedN* way to general type:

$$curriedN : \forall\alpha.\, [\mathbb{N}] \to \alpha$$

and don't type check its implementation. The *curried2* and *curried3* can still be successfully type checked.

Here we make a choice telling type checker to trust us, giving the function too general type. But in some cases, we give too restrictive type. For example *pluck* function can be given a correct structural type. However as it is used only on (heterogenic) arrays of arbitraries, with only three different "field" names, we can give it a type to cover these cases only:

$$\begin{aligned} pluck : \forall\alpha s.\, &[Map\ Arbitrary\ \alpha s, \mathcal{S}\ \texttt{"generator"}] \to Map\ Generator\ \alpha s \\ &\cap [Map\ Arbitrary\ \alpha s, \mathcal{S}\ \texttt{"shrink"}] \to Map\ Shrink\ \alpha s \\ &\cap [Map\ Arbitrary\ \alpha s, \mathcal{S}\ \texttt{"show"}] \to Map\ Show\ \alpha s \end{aligned}$$

The $\mathcal{S}$"`generator`" stands for singleton, "literal" string:

```
var gens = pluck(arbs, "generator");
```

where we pick the first intersection branch of the *pluck* type, as a literal `"generator"` is the second argument.

5.4 Type system sketch

In this section we'll summarise the special features of JSVERIFY described in the previous sections into single coherent presentation. We'd like to have a type system with

KINDS	κ	$:= \star$	*monotype*
		$\mid [\star]$	*typelist*
TYPESTRINGS	s	$:=$ `"str",...`	*type level string*
LITERALS	l	$:= Bool, String, \ldots$	*nullary types*, $\star$
	F	$:= List, Promise, \ldots$	*unary types*, $\star \to \star$
MONOTYPES	τ	$:= \alpha$	*type variable*
		$\mid \tau s \to \tau$	*multi-argument function*
		$\mid \tau \cap \tau'$	*disjoint intersection*
		$\mid l$	*literal*
		$\mid F\ \tau$	*type application*
		$\mid \mathcal{S}\ s$	*singleton string*
TYPELIST	τs	$:= []$	*empty list*
		$\mid \tau ::: \tau s$	*cons*
		$\mid \tau s :::^c \tau$	*snoc*
		$\mid Map\ F\ \tau s$	*type map*
POLYTYPES	σ	$:= \tau$	*monotype*
		$\mid \forall \alpha . \sigma$	*universal quantification*, $\star$
		$\mid \forall \alpha s . \sigma$	*universal quantification*, $[\star]$

Figure 5.1: TYPESSON type syntax

- Polymorphic types
- Type level lists
- Multi-argument functions
- Intersection types
- Built-in nominal types
- and literals

By combining all of these requirements, we get syntactically larger type language than previous ones. The syntax is shown in Figure 5.1. However, there aren't much more typing rules than before, as seen in Figure 5.2. As before, the constraint language can be read directly from typing rules, there

$$\frac{\sigma \in \Gamma \quad \sigma \preceq \tau}{\Gamma \vdash \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash \sigma \quad \alpha : \kappa \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \forall \alpha.\, \sigma}\ \textsc{Gen}$$

$$\frac{\Gamma, \tau s \vdash \tau \quad F \in \mathcal{F}}{\Gamma \vdash \tau s \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash \tau s \to \tau \quad \Gamma \vdash \tau s}{\Gamma \vdash \tau}\ \textsc{App}$$

$$\frac{}{\varepsilon \vdash []}\ \textsc{Nil} \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau s}{\Gamma \vdash \tau ::: \tau s}\ \textsc{Snoc} \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau s}{\Gamma \vdash \tau s :::^{c} \tau}\ \textsc{Snoc}$$

$$\frac{}{\Gamma \vdash \mathit{Map}\ F\ []}\ \textsc{Map-Nil} \qquad \frac{\Gamma \vdash F\ \tau \quad \Gamma \vdash \mathit{Map}\ F\ \tau s}{\Gamma \vdash \mathit{Map}\ F\ (\tau ::: \tau s)}\ \textsc{Map-Cons}$$

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta \quad \alpha \uplus \beta}{\Gamma \vdash \alpha \cap \beta}\ \textsc{Intersection}$$

$$\frac{\Gamma \vdash x : \alpha \cap \beta}{\Gamma \vdash x : \alpha}\ \textsc{Fst} \qquad \frac{\Gamma \vdash x : \alpha \cap \beta}{\Gamma \vdash x : \beta}\ \textsc{Snd}$$

$$\frac{}{\Gamma \vdash \texttt{"foo"} : \mathit{String}}\ \textsc{String} \qquad \frac{}{\Gamma \vdash \texttt{"foo"} : \mathcal{S}\ \texttt{"foo"}}\ \textsc{SingStr}$$

Figure 5.2: TYPESSON type system

are $\preceq$ and $\uplus$ relation based constraints, in addiction to the type-equality, $\equiv$. Also there is a choice in a constraint solver, as typing rules aren't entirely syntax-driven. There is a AGDA formalisation of TYPESSON type-system Appendix B.5, with embedding functions. In the following subsections, we'll explain the rules, and see some examples.

5.4.1 Basics

We copy two rules, VAR and GEN directly from the syntax-directed Hindley-Milner type-system (Section 3.3.3). We don't need arbitrary rank types in TYPESSON, so HM was a simple choice.

$$\frac{\sigma \in \Gamma \quad \sigma \preceq \tau}{\Gamma \vdash \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash \sigma \quad \alpha : \kappa \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \forall \alpha.\, \sigma}\ \textsc{Gen}$$

The $\preceq$ relation in VAR rule instantiates both kinds of variables, $\star$ and $[\star]$, the extension of the relation is straight-forward. There aren't let-expressions in the syntax language, so it doesn't make sense to have LET rule. We use GEN rule, to generalise top-level definitions (over the both kinds), i.e. the generalisation happens outside of the constraint solving. We have the same

syntax when abstracting over $\star$ or $[\star]$, which let us have single rule. More pedantically, we could write $\forall(a : \kappa).\, \sigma$, but we omit the kind if there aren't possibility for a confusion.

We don't reuse HM ABS and APP rules, as we'll need to modify them a little, which we'll do in the next section.

5.4.2 Functions

As mentioned before, we want to have multi-argument functions. The type-level lists are closely related to this need. We implement multi-argument function rules by starting with single argument function rules, as in other systems, but abstracting over a heterogeneous lists.

$$\frac{\Gamma, \tau s \vdash \tau \qquad F \in \mathcal{F}}{\Gamma \vdash \tau s \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash \tau s \to \tau \qquad \Gamma \vdash \tau s}{\Gamma \vdash \tau}\ \textsc{App}$$

where $\mathcal{F}$ is a collection of various type constructors, which we'll discuss properly in Section 5.4.4.

The rules are essentially the same as in the type system described previously, except that we abstract over a list of arguments. There is a few rules operating on $[\star]$, also, to construct and map over heterogeneous lists:

$$\frac{}{\varepsilon \vdash []}\ \textsc{Nil} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau s}{\Gamma \vdash \tau ::: \tau s}\ \textsc{Snoc} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau s}{\Gamma \vdash \tau s :::^{c} \tau}\ \textsc{Snoc}$$

$$\frac{}{\Gamma \vdash \mathit{Map}\ F\ []}\ \textsc{Map-Nil} \qquad \frac{\Gamma \vdash F\ \tau \qquad \Gamma \vdash \mathit{Map}\ F\ \tau s}{\Gamma \vdash \mathit{Map}\ F\ (\tau ::: \tau s)}\ \textsc{Map-Cons}$$

The following example will show how we use these rules.

Example 5.3 (N-ary products)**.** Recall a *zipWith* function in Section 5.3.2

$$\mathit{zipWith} : \forall \alpha s\ \beta.\, [\ldots \mathit{Map}\ \mathit{List}\ \alpha s, \mathit{FoldFun}\ \alpha s\ \beta] \to \mathit{List}\ \beta$$

Using our current type language, its type looks like:

$$\mathit{zipWith} : \forall(\alpha s : [\star], \beta : \star).\, \mathit{Map}\ \mathit{List}\ \alpha s :::^{r} (\alpha s \to \beta) \to \mathit{List}\ \beta$$

where $:::^{r}$ is a "snoc" operator, i.e. appending element to the list.

If we have two lists $xs, ys : \mathit{List}\ \mathbb{N}$ and we zip them with $\mathit{plus} : [\mathbb{N}, \mathbb{N}] \to \mathbb{N}$, to get $\mathit{List}\ \mathbb{N}$:

$$zipWith(xs, ys, plus)\ \ //\ : List\ \mathbb{N}$$

From the call-site, we can gather a constraints for *zipList* argument list:

$$\begin{aligned}\tau s &\equiv [\![xs]\!] ::: [\![ys]\!] ::: [\![plus]\!] ::: [] \\ &\equiv List\ \mathbb{N} ::: List\ \mathbb{N} ::: ([\mathbb{N}, \mathbb{N}] \to \mathbb{N}) ::: [] \\ &\equiv [List\ \mathbb{N}, List\ \mathbb{N}, [\mathbb{N}, \mathbb{N}] \to \mathbb{N}]\end{aligned}$$

Then, at some point, constraint solver will try to unify that type with instantiated type of the arguments of *zipList*:

$$[List\ \mathbb{N}, List\ \mathbb{N}, [\mathbb{N}, \mathbb{N}] \to \mathbb{N}] \equiv Map\ List\ \alpha s :::^r\ (\alpha s \to \beta) \to List\ \beta$$

Constraint solver is aware of lists, and can deduce the substitution:

$$\begin{aligned}\alpha s &\longmapsto [\mathbb{N}, \mathbb{N}] \\ \beta &\longmapsto \mathbb{N}\end{aligned}$$

thus accept the *zipList* expression above.

We should point out, that if the type lists are formed only by the cons operation, :::, then the basic unification would be enough to solve the type list equality constraints. However, in the presence of snoc, $:::^r$ and *Map* operations, the constraint solver have to be aware of the lists' structure. For example

$$\forall (x : \star)\ (xs : [\star]), \exists (y : \star)\ (ys : [\star]), x ::: xs \equiv ys :::^r y$$

equality is needed to deduce the substiution above.

Variadic functions complicate system quite a bit. They don't bring any additional expressive power into the type language, which wouldn't be possible using intersections, which we discuss next. However, the type language is more close to the term language, which greatly simplifies both constraint generation, and to some extend, the constraint solver.

5.4.3 Intersection types

Intersections are useful when giving types to ad-hoc polymorphic functions. The typing-rules resemble the conjunction rules from the propositional

logic, however the terms look different.

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta \quad \alpha \uplus \beta}{\Gamma \vdash \alpha \cap \beta}\ \textsc{Intersection}$$

$$\frac{\Gamma \vdash x : \alpha \cap \beta}{\Gamma \vdash x : \alpha}\ \textsc{Fst} \qquad \frac{\Gamma \vdash x : \alpha \cap \beta}{\Gamma \vdash x : \beta}\ \textsc{Snd}$$

The $\uplus$ relation indicates that the two types should be distinct, for example $\mathbb{N} \uplus \mathit{String}$, $\alpha \uplus \mathbb{N}$, and $\mathbb{N} \uplus \mathbb{N}$. An equivalent formulation, is that types should be non-unifiable, $\alpha \uplus \beta \Leftrightarrow \alpha \not\equiv \beta$. This is something we can decide as we have a closed type universe. There is also a paper accepted to the ICFP 2016 by Bruno Oliveira *et al.* [55] presents disjoint intersection types, the approach though is very similar, still differs slightly, as we don't introduce subtyping relation.

Example 5.4 (Intersection)**.** In Section 5.3.4 we mentioned the *adhoc* function. Let's prove that

$$\mathit{adhoc}\ 10.0 : \mathit{Number}$$

The literal 10.0 is obviously a *Number*. The rest of the proof follows trivially: We need to use $\uplus E_1$ rule, as even the $\uplus E_2$ is applicable, we'll be stuck after it.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathit{adhoc} : [\mathit{Number}] \to \mathit{Number} \cap \ldots}\ \textsc{Var}}{\Gamma \vdash \mathit{adhoc} : [\mathit{Number}] \to \mathit{Number}}\ \textsc{Fst} \qquad \dfrac{}{\Gamma \vdash 10.0 : \mathit{Number}}\ \textsc{Number}}{\Gamma \vdash \mathit{adhoc}\ 10.0 : \mathit{Number}}\ \textsc{App}$$

It's possible that someone would write a function, where type variable is free, leading to the situation:

$$\Gamma, x : \alpha \vdash \mathit{adhoc}\ x : \alpha$$

We reject such definitions. Because during the constraint solving process, α would unify with *Number* or *String*.

When type-checking a function with an intersection type, we

- Check that intersection halves are distinct, and
- Type-check function specialised to the each half of the intersection.

5.4.4 Nominal types

The second last part of the TYPESSON type system are the nominal types. There is a handful of types:

$$\begin{aligned}\mathcal{L} &= \{Number, Bool, String, \ldots\} \\ \mathcal{F} &= \{List, LazySeq, Arbitrary, Generator, Shrink, Show, \\ &\quad Promise, Trampoline, NotPromise, \ldots\}\end{aligned}$$

Most of these types we have already encountered. The first ones are standard JAVASCRIPT types. Then we have truly JSVERIFY domain specific types. The rest are types from the libraries, we can briefly discuss:

The *LazySeq* type is a type of lazy sequences. We use it in shrinking, as lazily producing shrinking results is important for the performance. There a few dozen functions provided in the separate library[6], which can be given types in TYPESSON system.

The *Generator* type is twofold. It's a opaque nominal type, but it can be also used as a function $[\mathbb{N}] \to \alpha$. Similar duplicity, is with *Shrink* and *Show*. Essentially, we have informal rule, that we can implicitly coerce these nominal types, into their function-forms, when looking for the type of the function. At the moment, this is very ad-hoc approach. Better way, is to introduce a sub-typing relation, which is used in the APP rule.

The *Promise* type is a special type[7] to work with asynchronous computations in JAVASCRIPT. The *Trampoline* is a special type to be able to write functions in chaining style, i.e. similar to promise *.then* chains, without overflowing the call-stack. The *NotPromise* type is everything else. Using these three types we can write functions on the computation wrappers, like

$$\begin{aligned}fmap : \forall \alpha\ \beta.\ &[Promise\ \alpha, \alpha \to NotPromise\ \beta] \to Promise\ \beta \\ \cap &[Trampoline\ \alpha, \alpha \to NotPromise\ \beta] \to Trampoline\ \beta \\ \cap &[Trampoline\ \alpha, \alpha \to NotPromise\ \beta] \to NotPromise\ \beta\end{aligned}$$

This is not as elegant as $fmap : Functor\ f \Rightarrow (\alpha \to \beta) \to f\ \alpha \to f\ \beta$ as in HASKELL, but we don't even try to be that general. It's worth noticing that, in JAVASCRIPT, you shouldn't be able construct value of a nested type $Promise\ (Promise\ \alpha)$, therefore we have *NotPromise* sprinkled into many of the function types.

[6] `https://www.npmjs.com/package/lazy-seq`
[7] `https://promisesaplus.com/`

Also we have functions with a more tricky and non-uniform types, for example[8]:

$$\begin{aligned} \mathit{pure} : \forall\alpha.\, [\mathit{Promise}\ \alpha] &\to \mathit{Promise}\ \alpha \\ \cap [\mathit{Trampoline}\ \alpha] &\to \mathit{Trampoline}\ \alpha \\ \cap [\mathit{NotPromise}\ \alpha] &\to \mathit{Trampoline}\ \alpha \end{aligned}$$

5.4.5 Literals

Literals are the last part of the TYPESSON type-system. Otherwise, they would be straightforward, as the expression "carries" its type, but we have a small complication: *singleton string types*.

$$\frac{}{\Gamma \vdash \texttt{"foo"} : \mathit{String}}\ \textsc{String} \qquad \frac{}{\Gamma \vdash \texttt{"foo"} : \mathcal{S}\ \texttt{"foo"}}\ \textsc{SingStr}$$

Singleton strings are used as tags. As they are different types, they are disjoint, and can be used to select different branches of intersections. See *pluck* example in Section 5.3.5.

There are also other literal types, for booleans and numbers, but we omitted them for brevity.

5.5 Conclusion

In this chapter we presented the TYPESSON framework, a type-system tailored for the JSVERIFY library written in JAVASCRIPT. We have positive experience with the results. There are few silly "forgot to bless" type of bugs which JSVERIFY successfully caught. For example the *shrinkTuple* function was missing the *shrinkBless* call:

```
function shrinkTuple(shrinks) {
    var shrink = // ...
    var result = shrinkBless(shrink);
    return utils.curried2(result, arguments);
}
```

The incomplete definition wasn't caught by tests, even JSVERIFY had a 100% coverage test from the beginning.

[8] We abuse the names on purpose.

We also feel more confident about correctness about the JSVERIFY main loop, where we operate on promises and trampolines. There are many nested *pure*, *fmap* and other similar functions, and tracking the types "manually" is quite difficult.

The TYPESSON story isn't complete, however. There are various features which could be added: *union* types to complement the *intersections*, also *records*, maybe using *row polymorphism*.

TYPESCRIPT[9] is a rapidly developed typed version of JAVASCRIPT, and has many features present in TYPESSON. For example there are *String Literal Types* and *Intersection Types*. On the other hand TYPESCRIPT lacks support for the heterogeneous functions, which was the major motivation to the development of TYPESSON in the first place.

Another possible development direction of TYPESSON is to provide a preprocessing mode. Such we can add dynamic checks for the types, especially in the functions in the public API. Currently we have manual *assert* not systematically added to the code base, they could be added automatically into *debug* build. Further, we can do some other type-directed transformations as well.

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

Chapter 6

Conclusion

In this thesis we have described two type-systems: TYPEDLAB for a subset of MATLAB, and TYPESSON for a subset of JAVASCRIPT. Both type-systems have advanced features: a constrained natural number indexing or intersection types and type level lists.

To ensure correctness of the type-systems, we show that they are embeddable into intuitionistic type theory as implemented in the AGDA programming language. We use a constraint solving based approach to implement the type-inference for the systems. The constraint language can be read from the rule specification, which in turn, are given "for free" if we can formulate the type system as a Pure Type System. Also we discuss how constraints can be solved: we explain unification process for the structual type equality constraints and mention various decision procedures for other decidable theories.

The implementations of TYPEDLAB and TYPESSON lack proper front-ends, which would make the type-checking process as easy as running a console command. Ideally, user could run `typesson file.js` command to type-check the file. However, some non-trivial manual steps are required at the moment. In other words, they aren't yet ready for the wider use. Finalising the implementation is one further development topic.

In the future, we would also like to extend either one of the type systems, or experiment with the approach to implement new domain specific typed languages. One interesting topic is *linear types* [11, 21, 68], which can be used to implement a specification language for concurrent processes. Another option is to develop a fully type-inferrable typed command line scripting language with heterogenous functions and regular expressions.

Bibliography

[1] ALMEIDA, M., MOREIRA, N., AND REIS, R. Testing the equivalence of regular languages.

[2] ALMEIDA, R., BRODA, S., AND MOREIRA, N. Deciding kat and hoare logic with derivatives.

[3] ARMSTRONG, J. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, 2006.

[4] AUGUSTSSON, L. Cayenne – a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 1998), ICFP '98, ACM, pp. 239–250.

[5] BARENDREGT, H., ABRAMSKY, S., GABBAY, D. M., MAIBAUM, T. S. E., AND BARENDREGT, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science* (1992), Oxford University Press, pp. 117–309.

[6] BARENDREGT, H. P. *The Lambda Calculus Its Syntax and Semantics*, revised ed., vol. 103. North Holland, 1984. http://www.cs.ru.nl/ henk/Personal Webpage.

[7] BERARDI, S. Towards a mathematical analysis of the coquand–huet calculus of constructions and the other systems of the barendregt cube. Tech. rep., Department of Computer Science, Carnegie-Mellon University and Dipartimento di Matematica, Universita di Torino, 1988.

[8] BEREZIN, S., GANESH, V., AND DILL, D. L. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2003), TACAS'03, Springer-Verlag, pp. 521–536.

[9] BONNAIRE-SERGEANT, A. A practical optional type system for clojure. Master's thesis, The University of Western Australia, 2012.

[10] BRZOZOWSKI, J. A. Derivatives of regular expressions. *J. ACM 11*, 4 (Oct. 1964), 481–494.

[11] CAIRES, L., AND PFENNING, F. *Session Types as Intuitionistic Linear Propositions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 222–236.

[12] CARDELLI, L. A polymorphic λ-calculus with type:type. Tech. Rep. 10, DEC Systems Research Center, 1986.

[13] CHUGH, R., HERMAN, D., AND JHALA, R. Dependent types for javascript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2012), OOPSLA '12, ACM, pp. 587–606.

[14] CLAESSEN, K., AND HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[15] CLÉMENT, D., DESPEYROUX, T., KAHN, G., AND DESPEYROUX, J. A simple applicative language: Mini-ml. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1986), LFP '86, ACM, pp. 13–27.

[16] COOPER, D. Theorem proving in arithmetic without multiplication. *Machine Intelligence 7* (1972), 91–99.

[17] COQUAND, T., AND HUET, G. The calculus of constructions. *Inf. Comput. 76*, 2-3 (Feb. 1988), 95–120.

[18] CURRY, H. B. The inconsistency of certain formal logic. *The Journal of Symbolic Logic 7*, 3 (1942), 115–117.

[19] DAMAS, L. M. M. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[20] DE VRIES, E., AND LÖH, A. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (New York, NY, USA, 2014), WGP '14, ACM, pp. 83–94.

[21] DE VRIES, E., PLASMEIJER, R., AND ABRAHAMSON, D. M. *Uniqueness Typing Simplified*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 201–218.

[22] DOCKINS, R. The GHC typechecker is Turing-complete. `https://mail.haskell.org/pipermail/haskell/2006-August/018355.html`, 2006. [Online; accessed 17-August-2016].

[23] DUERIG, M. Scala type level encoding of the SKI calculus. `https://michid.wordpress.com/2010/01/29/scala-type-level-encoding-of-the-ski-calculus/`, 2011. [Online; accessed 17-August-2016].

[24] ECMAScript® 2016 Language Specification. Standard ECMA-262, Ecma International, June 2016.

[25] EISENBERG, R. A. An overabundance of equality: Implementing kind equalities into haskell. Tech. Rep. MS-CIS-15-10, University of Pennsylvania, 2015.

[26] FACEBOOK. About Flow. `http://flowtype.org/docs/about-flow.html#_`, 2016. [Online; accessed 27-April-2016].

[27] FACEBOOK. Hack overview. `https://docs.hhvm.com/hack/overview/typing`, 2016. [Online; accessed 27-April-2016].

[28] FENTON, S. *TypeScript For JavaScript Programmers*. Lulu.com, 2012.

[29] FLANAGAN, D. *JavaScript - The Definitive Guide*, 6 ed. O'Reilly, 2011.

[30] GEUVERS, H. Induction is not derivable in second order dependent type theory.

[31] GIRARD, J.-Y. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7., 1972. in French.

[32] GREGOIRE, B., AND MAHBOUBI, A. Proving equalities in a commutative ring done right in coq. In *Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS 3603* (2005), Springer, pp. 98–113.

[33] GUNDRY, A. M. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[34] GUNDRY, A. M. A typechecker plugin for units of measure: Domain-specific constraint solving in ghc haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (New York, NY, USA, 2015), Haskell '15, ACM, pp. 11–22.

[35] HEEREN, B., HAGE, J., AND SWIERSTRA, S. D. Constraint based type inferencing in helium. In *Workshop Proceedings of Immediate Applications of Constraint Programming* (Cork, 2003), pp. 59–80.

[36] HEEREN, B., HAGE, J., AND SWIERSTRA, S. D. Scripting the type inference process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2003), ICFP '03, ACM, pp. 3–13.

[37] HINDLEY, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (1969), 29–60.

[38] HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[39] HUDAK, P., PEYTON JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMÁN, M. M., HAMMOND, K., HUGHES, J., JOHNSSON, T., KIEBURTZ, D., NIKHIL, R., PARTAIN, W., AND PETERSON, J. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not. 27*, 5 (May 1992), 1–164.

[40] HURKENS, A. J. C. *A simplification of Girard's paradox*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 266–278.

[41] IERUSALIMSCHY, R. *Programming in Lua*. Lua.Org, Aug. 2016.

[42] JSVERIFY. JSVerify homepage. `http://jsverify.github.io/`, 2015. [Online; accessed 27-April-2016].

[43] KENNEDY, A. J. Relational parametricity and units of measure. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 442–455.

[44] LÄUFER, K., AND ODERSKY, M. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994), 1411–1430.

[45] LEIVANT, D. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1983), SFCS '83, IEEE Computer Society, pp. 460–469.

[46] MARLOW, S. Haskell 2010: Language report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010. [Online; accessed 27-April-2016].

[47] MARTELLI, A., AND MONTANARI, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 258–282.

[48] MARTIN-LÖF, P. An intuitionistic theory of types, 1972.

[49] MARTIN-LÖF, P. An intuitionistic theory of types: Predicative part. In *Logic Proceedings of the Logic Colloquim* (1975), Logic colloquium 1973, pp. 73–118.

[50] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[51] MOLER, C. The Origins of MATLAB. `http://se.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html`, 2004. [Online; accessed 26-April-2016].

[52] NANEVSKI, A., PFENNING, F., AND PIENTKA, B. Contextual modal type theory. *ACM Trans. Comput. Logic 9*, 3 (June 2008), 23:1–23:49.

[53] NORDSTRÖM, B., PETERSSON, K., AND SMITH, J. M. *Programming in Martin-Löf's Type Theory: An Introduction*, 0 ed. Oxford University Press, USA, July 1990.

[54] NORELL, U. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[55] OLIVEIRA, B. C. D. S., SHI, Z., AND ALPUIM, J. A. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2016), ICFP 2016, ACM, pp. 364–377.

[56] OWENS, S., REPPY, J., AND TURON, A. Regular-expression derivatives re-examined. *J. Funct. Program. 19*, 2 (Mar. 2009), 173–190.

[57] PEYTON JONES, S., VYTINIOTIS, D., WEIRICH, S., AND SHIELDS, M. Practical type inference for arbitrary-rank types. *J. Funct. Program. 17*, 1 (Jan. 2007), 1–82.

[58] PFENNING, F., AND DAVIES, R. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci. 11*, 4 (Aug. 2001), 511–540.

[59] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[60] PIERCE, B. C. Software foundations. `https://www.cis.upenn.edu/~bcpierce/sf/current/index.html`, 2016. [Online].

[61] PRESBURGER, M. Ueber die vollstaendigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrés de Mathématiciens des Pays Slaves*. Warsaw, Poland, 1929, pp. 92–101.

[62] PUGH, W. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 1991), Supercomputing '91, ACM, pp. 4–13.

[63] QUVIQ. Erlang QuickCheck. `http://www.quviq.com/products/erlang-quickcheck/`, 2016. [Online; accessed 1-September-2016].

[64] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.

[65] ROSSER, B. An informal exposition of proofs of gödel's theorems and church's theorem. *Journal of Symbolic Logic 4* (6 1939), 53–60.

[66] TERLOUW, J. Een nadere bewijstheoretische analyse van gstt's. Tech. rep., Department of Computer Science, Catholic University of Nijmegen, 1989. in Dutch.

[67] TOBIN-HOCHSTADT, S., AND FELLEISEN, M. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008), POPL '08, ACM, pp. 395–406.

[68] TONINHO, B. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Universidade Nova de Lisboa, May 2015.

[69] VELDHUIZEN, T. L. C++ templates are turing complete. Tech. rep., 2003.

[70] VYTINIOTIS, D., PEYTON JONES, S., AND SCHRIJVERS, T. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2010), TLDI '10, ACM, pp. 39–50.

[71] WEIRICH, S., HSU, J., AND EISENBERG, R. A. System fc with explicit kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2013), ICFP '13, ACM, pp. 275–286.

[72] WELLS, J. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic 98*, 1 (1999), 111 – 156.

[73] WIIK, J., AND BOSTRÖM, P. Contract-based verification of matlab-style matrix programs. *Formal Aspects of Computing 28*, 1 (2016), 79–107.

Appendix A

Judgements

A.1 Type judgment $\vdash and : Bool \to Bool \to Bool$ in System F

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : Bool, y : Bool \vdash x : \forall\alpha.\, \alpha \to \alpha \to \alpha}\ \textsc{Weaken + Start}}{x : Bool, y : Bool \vdash x\ Bool : Bool \to Bool \to Bool}\ \textsc{Inst} \qquad \dfrac{\vdots}{\cdots, y : Bool \vdash y : Bool}}{x : Bool, y : Bool \vdash x\ Bool\ y : Bool \to Bool}\ \textsc{App} \qquad \dfrac{\vdots}{\cdots \vdash false : Bool}}{x : Bool, y : Bool \vdash x\ Bool\ y\ false : Bool}\ \textsc{App}$$

$$\dfrac{x : Bool \vdash \lambda y : Bool.\, x\ Bool\ y\ false : Bool \to Bool}{\varepsilon \vdash \lambda x : Bool.\, \lambda y : Bool.\, x\ Bool\ y\ false : Bool \to Bool \to Bool}\ \textsc{Gen}$$

$$\dfrac{x : Bool, y : Bool \vdash x\ Bool\ y\ false : Bool}{x : Bool \vdash \lambda y : Bool.\, x\ Bool\ y\ false : Bool \to Bool}\ \textsc{Gen}$$

A.2 Type judgment $\vdash$ *true* : *Bool* in PTS System F

First recall the derivation of $\varepsilon \vdash \mathit{Bool} : \star$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha : \star \vdash \alpha : \star}}{\alpha : \star, t : \alpha \vdash \alpha : \star}}{\alpha : \star, t : \alpha, f : \alpha \vdash \alpha : \star}}{\alpha : \star, t : \alpha \vdash \Pi f : \alpha.\, \alpha : \star}}{\alpha : \star \vdash \Pi t : \alpha.\, \Pi f : \alpha.\, \alpha : \star}}{\varepsilon \vdash \star : \square}}{\varepsilon \vdash \Pi \alpha : \star.\, \Pi t : \alpha.\, \Pi f : \alpha.\, \alpha : \star} \tag{A.1}$$

There is also a large part which we will reuse:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha : \star \vdash \alpha : \star}}{\alpha : \star, t : \alpha \vdash \alpha : \star}}{\alpha : \star, t : \alpha, f : \alpha \vdash \alpha : \star}}{\alpha : \star, t : \alpha \vdash \Pi f : \alpha.\, \alpha : \star}}{\alpha : \star \vdash \Pi t : \alpha.\, \Pi f : \alpha.\, \alpha : \star}}{} \tag{A.2}$$

And finally $\varepsilon \vdash \mathit{true} : \mathit{Bool}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha : \star, t : \alpha, f : \alpha \vdash t : \alpha}}{\alpha : \star, t : \alpha \vdash \lambda f : \alpha.\, t : \Pi f : \alpha.\, \alpha}}{\alpha : \star \vdash \lambda t : \alpha.\, \lambda f : \alpha.\, t : \Pi t : \alpha.\, \Pi f : \alpha.\, \alpha}}{\varepsilon \vdash \lambda a : \star.\, \lambda t : \alpha.\, \lambda f : \alpha.\, t : \Pi \alpha : \star.\, \Pi t : \alpha.\, \Pi f : \alpha.\, \alpha} \tag{A.3}$$

Appendix B

Code listings

B.1 Nu implies falsehood

If we try to define *Nu* as

```
data Nu (a : Set) : Set where
  MkNu : (Nu a → a) → Nu a
```

we get an error:

```
Nu is not strictly positive, because it occurs
to the left of an arrow
in the type of the constructor MkNu
in the definition of Nu.
```

It's not hard to show, that if such type existed, we can derive falsehood:

```
data ⊥ : Set where

absurd : {a : Set} → ⊥ → a
absurd ()

record NuDef : Set₁ where
  constructor mkNuDef
  field
    Nu : Set → Set
    mkNu : {a : Set} → (Nu a → a) → Nu a
    unNu : {a : Set} → Nu a → (Nu a → a)

nu-unsound : NuDef → ⊥
```

```
nu-unsound (mkNuDef Nu mkNu unNu) = unNu' (unNu' nu-nu)
  where
    – we can unwrap Nu:
  unNu' : {a : Set} → Nu a → a
  unNu' nu = unNu nu nu
    – We can create double-Nu from a thin air
  nu-nu : {a : Set} → Nu (Nu a)
  nu-nu = mkNu (λ nn → unNu nn nn)
record NuC (a : Set) : Set₁ where
  constructor mkNuC
  field nuc : {b : Set} → (b → a) → b
nuc-⊥ : {a : Set} → NuC a → ⊥
nuc-⊥ (mkNuC nuc) = nuc (λ x → absurd x)
```

B.2 STLC embedded into Agda

```
infixr 7 _⇒_
data Type : Set where
  nat  : Type
  _⇒_ : (a b : Type) → Type
Cxt = List Type
data Term (Γ : Cxt) : Type → Set where
  var : ∀ {a} (i : a ∈ Γ) → Term Γ a
  lit : ℕ → Term Γ nat
  suc : Term Γ (nat ⇒ nat)
  app : ∀ {a b} → Term Γ (a ⇒ b) → Term Γ a → Term Γ b
  lam : ∀ {b} (a : Type) → Term (a :: Γ) b → Term Γ (a ⇒ b)
El : Type → Set
El nat     = ℕ
El (a ⇒ b) = El a → El b
Env : Cxt → Set
Env Γ = NP El Γ
eval : ∀ {Γ a} → Term Γ a → Env Γ → El a
eval (var i) e = lookup∈ e i
eval (lit x) e = x
```

```
eval suc e          = suc
eval (app t t₁) e   = eval t e (eval t₁ e)
eval (lam a t) e y  = eval t (y :: e)
```

B.3 TypedLab1

```
data Index : ℕ → Set where
  IndexConst : {n : ℕ} → ℕ → Index n
  IndexVar   : {n : ℕ} → Fin n → Index n

data Type : ℕ → Set where
  TypeM   : {n : ℕ} → Index n → Index n → Type n
  TypeArr : {n : ℕ} → Type n → Type n → Type n
  TypeAbs : {n : ℕ} → Type (suc n) → Type n

postulate
  M : ℕ → ℕ → Set

fromIndex : ∀ {n} → Vec ℕ n → Index n → ℕ
fromIndex _ (IndexConst n) = n
fromIndex v (IndexVar i)   = lookup i v

fromType' : ∀ {n} → Vec ℕ n → (m : Type n) → Set
fromType' v (TypeM i j)   = M (fromIndex v i) (fromIndex v j)
fromType' v (TypeArr a b) = fromType' v a → fromType' v b
fromType' v (TypeAbs t)   = (m : _) → fromType' (m :: v) t

fromType : ∀ {n} → Type n → Set
fromType {zero} x  = fromType' [] x
fromType {suc n} x = fromType (TypeAbs x)

{- Type of trace: ∀n. 𝕄 n n → 𝕄 1 1. -}
traceType : Type 0
traceType = TypeAbs (TypeArr
  (TypeM (IndexVar zero) (IndexVar zero))
  (TypeM (IndexConst 1) (IndexConst 1)))

{- Type of trace in AGDA -}
traceTypeAgda : Set
traceTypeAgda = (n : ℕ) → M n n → M 1 1

{- Example, trivial proof that fromType maps the type correctly -}
traceTypeProof : fromType traceType ≡ traceTypeAgda
traceTypeProof = refl
```

B.4 TypedLab2

```
data Index : ℕ → Set where
  IndexConst : {n : ℕ} → ℕ → Index n
  IndexVar   : {n : ℕ} → Fin n → Index n
data Constraint : ℕ → Set where
  ConstraintLe : {n : ℕ} → Fin n → Fin n → Constraint n
data Type : ℕ → Set where
  TypeM    : {n : ℕ} → Index n → Index n → Type n
  TypeArr  : {n : ℕ} → Type n → Type n → Type n
  TypeAbs  : {n : ℕ} → Type (suc n) → Type n
  TypeCons : {n : ℕ} → Constraint n → Type n → Type n
postulate
  M : ℕ → ℕ → Set
fromIndex : ∀ {n} → Vec ℕ n → Index n → ℕ
fromIndex _ (IndexConst n) = n
fromIndex v (IndexVar i)   = lookup i v
fromConstraint : ∀ {n} → Vec ℕ n → Constraint n → Set
fromConstraint v (ConstraintLe i j) = lookup i v ≤ lookup j v
fromType' : ∀ {n} → Vec ℕ n → (m : Type n) → Set
fromType' v (TypeM i j)     = M (fromIndex v i) (fromIndex v j)
fromType' v (TypeArr a b)   = fromType' v a → fromType' v b
fromType' v (TypeAbs t)     = (m : _) → fromType' (m :: v) t
fromType' v (TypeCons c t)  = fromConstraint v c → fromType' v t
fromType : ∀ {n} → Type n → Set
fromType {zero} x  = fromType' [] x
fromType {suc n} x = fromType (TypeAbs x)
{- Type of leastsq: ∀n. ℕ n n → ℕ 1 1. -}
leastsqType : Type 0
leastsqType = TypeAbs (TypeAbs
    (TypeCons c (TypeArr mmn (TypeArr mn1 mn1))))
  where
    c   = ConstraintLe one zero
    mmn = TypeM (IndexVar one) (IndexVar zero)
    mn1 = TypeM (IndexVar zero) (IndexConst 1)
```

```
{- Type of leastsq in AGDA -}
leastsqTypeAgda : Set
leastsqTypeAgda =
    (m n : ℕ) → m ≤ n → M m n → M n 1 → M n 1
{- Example, trivial proof that fromType maps the type correctly -}
leastsqTypeProof : fromType leastsqType ≡ leastsqTypeAgda
leastsqTypeProof = refl
```

B.5 Typesson

Data definitions follow directly from the syntax described in Figure 5.1. The contexts, Δ are indexed by list of variable kinds.

We omit the definition of polytypes. It's very straighforward addition, and doesn't show anything interesting.

```
data Kind : Set where
  ⋆   : Kind
  [⋆] : Kind

Ctx : Set
Ctx = List Kind

ε : Ctx
ε = []

data Nullary : Set where
  nullary-ℕ      : Nullary
  nullary-Bool   : Nullary
  nullary-String : Nullary

data Unary : Set where
  unary-List : Unary

data Mono (Δ : Ctx) : Kind → Set where
  mono-var      : {k : Kind} → k ∈ Δ → Mono Δ k
  mono-nullary  : Nullary → Mono Δ ⋆
  mono-unary    : Unary → Mono Δ ⋆ → Mono Δ ⋆
  mono-sing-str : String → Mono Δ ⋆
  _⇒_           : Mono Δ [⋆] → Mono Δ ⋆ → Mono Δ ⋆
  mono-union    : Mono Δ ⋆ → Mono Δ ⋆ → Mono Δ ⋆
  mono-nil      : Mono Δ [⋆]
```

```
_::_          : Mono Δ ⋆ → Mono Δ [⋆] → Mono Δ [⋆]
mono-map      : Unary → Mono Δ [⋆] → Mono Δ [⋆]
```

Let's also define smart constuctors, for the ℕ type, and the unary function:

```
mono-ℕ : ∀ {Δ} → Mono Δ ⋆
mono-ℕ = mono-nullary nullary-ℕ

mono-arr : ∀ {Δ} → Mono Δ ⋆ → Mono Δ ⋆ → Mono Δ ⋆
mono-arr a b = (a :: mono-nil) ⇒ b
```

The most intersting part is the flattening of *Mono* value into AGDA type. The procedure is a structural recursion as in previous similar examples. We separate the type-variable substitution *subst*-mono and the embedding of closed type *Mono-ε→*Set. We also expand multi-argument functions into normal AGDA functions using the auxiliary *fold*-fun function, in a more precise formalisation, we'll need to actually use *NP*.

```
Env : Ctx → Set
Env Δ = NP (Mono ε) Δ

data SingString (s : String) : Set where
  ss : SingString s

Kind→Set₁ : Kind → Set₁
Kind→Set₁ ⋆   = Set
Kind→Set₁ [⋆] = List Set

Nullary→Set : Nullary → Set
Nullary→Set nullary-ℕ      = ℕ
Nullary→Set nullary-Bool   = Bool
Nullary→Set nullary-String = String

Unary→Set : Unary → Set → Set
Unary→Set tc a = List a

fold-fun : List Set → Set → Set
fold-fun []       b = b
fold-fun (a :: as) b = a → fold-fun as b

subst-mono : ∀ {Δ k} → Env Δ → Mono Δ k → Mono ε k
subst-mono e (mono-var i)
  = lookup∈ e i
subst-mono e (mono-nullary x)
  = mono-nullary x
subst-mono e (mono-unary f x)
```

```
    = mono-unary f (subst-mono e x)
subst-mono e (mono-sing-str s)
    = mono-sing-str s
subst-mono e (as ⇒ b)
    = subst-mono e as ⇒ subst-mono e b
subst-mono e (mono-union a b)
    = mono-union (subst-mono e a) (subst-mono e b)
subst-mono e mono-nil
    = mono-nil
subst-mono e (h :: t)
    = subst-mono e h :: subst-mono e t
subst-mono e (mono-map f as)
    = mono-map f (subst-mono e as)
Mono-ε→Set : ∀ {k} → Mono ε k → Kind→Set₁ k
Mono-ε→Set (mono-var ())
Mono-ε→Set (mono-nullary x)
    = Nullary→Set x
Mono-ε→Set (mono-unary f x)
    = Unary→Set f (Mono-ε→Set x)
Mono-ε→Set (mono-sing-str s)
    = SingString s
Mono-ε→Set (as ⇒ b)
    = fold-fun (Mono-ε→Set as) (Mono-ε→Set b)
Mono-ε→Set (mono-union a b)
    = Mono-ε→Set a × Mono-ε→Set b
Mono-ε→Set mono-nil
    = []
Mono-ε→Set (h :: t)
    = Mono-ε→Set h :: Mono-ε→Set t
Mono-ε→Set (mono-map f as)
    = L.map (Unary→Set f) (Mono-ε→Set as)
Mono→Set : ∀ {k Δ} → Env Δ → Mono Δ k → Kind→Set₁ k
Mono→Set e m = Mono-ε→Set (subst-mono e m)
```

Example B.1 (zipWith type)**.** We can encode the monotype part of the *zipWith*. Given simple environment, it has an expected AGDA type. We use cons instead of snoc, as we omitted snoc from this formalisation.

```
zipWith-Mono : Mono (⋆ :: [⋆] :: []) ⋆
zipWith-Mono = (f :: as) ⇒ mono-unary unary-List b
```

```
  where
    lists : Mono (⋆ :: [⋆] :: []) [⋆]
    lists = mono-var (suc zero)

    b : Mono (⋆ :: [⋆] :: []) ⋆
    b = mono-var zero

    f : Mono (⋆ :: [⋆] :: []) ⋆
    f = lists ⇒ b

    as : Mono (⋆ :: [⋆] :: []) [⋆]
    as = mono-map unary-List lists

zipWith-Env : Env (⋆ :: [⋆] :: [])
zipWith-Env = mono-ℕ :: (mono-ℕ :: mono-ℕ :: mono-nil) :: []

zipWith-Agda : Set
zipWith-Agda = (ℕ → ℕ → ℕ) → List ℕ → List ℕ → List ℕ

zipWith-Agda′ : Set
zipWith-Agda′ = Mono→Set zipWith-Env zipWith-Mono

zipWith-Example : zipWith-Agda′ ≡ zipWith-Agda
zipWith-Example = refl
```