Pure Type Systems for Functional Programming

Jan-Willem Roorda
jw@cs.uu.nl

Abstract

We present a functional programming language based on Pure Type Systems (PTSs). We show how we can define such a language by extending the PTS framework with algebraic data types, case expressions and definitions. To be able to experiment with our language we present an implementation of a type checker and an interpreter for our language.

PTSs are well suited as a basis for a functional programming language because they are at the top of a hierarchy of increasingly stronger type systems. The concepts of 'existential types', 'rank-n polymorphism' and 'dependent types' arise naturally in functional programming languages based on the systems in this hierarchy. There is no need for ad-hoc extensions to incorporate these features.

The type system of our language is more powerful than the Hindley-Milner system. We illustrate this fact by giving a number of meaningful programs that cannot be typed in Haskell but are typable in our language. A 'real world' example of such a program is the mapping of a specialisation of a Generic Haskell function to a Haskell function.

Unlike the description of the Henk language by Simon Peyton Jones and Erik Meijer we give a complete formal definition of the type system and the operational semantics of our language. Another difference between Henk and our language is that our language is defined for a large class of Pure Type Systems instead of only for the systems of the λ-cube.

Contents

Chapter 1

Introduction

One of the great advantages of functional programming languages is that they posses powerful type systems that prevent programmers from writing erroneous programs. If a program contains type errors the programmer is warned by a type checking algorithm before running the program. If a program is type error free, it is assumed to be safe to run: 'typed programs cannot go wrong' [17].

So, one of the requirements for a type system is that 'typed programs cannot go wrong', but preferably the implication should work the other way too, that is: 'good programs can be typed'. Unfortunately not all type systems adhere to this principle. For instance, in the functional programming language Haskell [11] it is easy to write programs that make perfect sense, but are rejected by the type checker. This has been been our main motivation to study more liberal type systems.

PTSs for Functional Programming: Why?

The Haskell programming language is based on the Hindley-Milner type system [17]. The Hindley-Milner system is an implicitly typed system. That means that we do not need -as with explicitly typed languages- to annotate variables in λ-abstractions with their type. A nice property of the Hindley-Milner system is that it combines implicit typing, polymorphism and automatic type inference. Unfortunately, the type system only allows a limited form of polymorphism. This implies that there exist Haskell programs that make perfect sense, but nevertheless are rejected by the type checker. An example of such a program is:

```
foo :: (a->a) -> Int
foo f = f (+) (f 2) (f 3)

five :: Int
five = foo id
```

Trying to run the above program by Hugs, a Haskell interpreter, yields a type error. The Haskell type system tries to fix a single monomorphic type for `f`. Because the program requires the type of `f` to be instantiated to both `Int->Int` and `(Int -> Int) -> (Int -> Int)` it will be rejected.

Admittedly, a slightly modified version of the above program can be typed by a recent extension[13] [16] of Hugs which allows rank-2 types. But also this extension does not provide everything we need: there are still meaningful programs left that cannot be typed.

A real world example of the limitations of the Haskell type system (with or without extensions) is the mapping of polytypic functions to Haskell programs. Polytypic functions are functions that are defined by induction on the structure of types. Using such functions it is for instance possible to define equality, pretty-printing and compression functions that work for all types. It is not possible to define polytypic functions in Haskell. It is, however, possible to write a mapping from a polytypic definition and a type to a Haskell function that is the specialisation of the polytypic function to that type. Unfortunately, some specialisations are mapped to Haskell functions that make perfect sense, but are rejected by the Haskell type checker [10, p. 6]. The reason for the rejection in this case is that the mapping requires universal quantification in the type at a place where Haskell does not allow such quantifiers to occur.

The above examples illustrate that the type system of Haskell (with or without extensions) is not strong enough for our current needs. In this thesis we investigate whether Pure Type Systems can solve the above problems and whether they can be used as a basis for a functional programming language. We have chosen to study the class of PTSs for its generality: PTSs generalise a large set of type systems (including the set of systems of the so called λ-cube). Furthermore, PTSs provide a single syntax for terms, types and kinds. This makes it possible to use a single data type to represent all three levels, and to use a single set of utility functions (like parsing, pretty-printing, substitution functions) that work on all levels. This leads to considerable code efficiency when writing tools (like compilers or interpreters) for languages based on PTSs.

PTSs for Functional Programming: How?

The idea to use PTSs as a basis for a functional programming language has first been described by by Erik Meijer and Simon Peyton Jones. In [12] they present a language called 'Henk'. Unfortunately, they only give typing rules for the λ-cube variant of 'Henk'. Furthermore, they only give a sketch of how to extend PTSs to a real language. The language presented in this paper is inspired by the work of Meijer and Jones, but we have chosen a more formal approach: we will give an precise definition of how to extend the theory of PTSs. Another difference between Henk and our language is that our language is defined for a large class of PTSs and not only for the systems of the λ-cube.

If we want to use the theory of PTSs as a basis for a functional programming language we have to investigate a couple of topics.

First of all we have to extend the PTS-framework: functional programming languages provide features, such as algebraic data types, case expressions and definitions, that are not in PTSs. To use PTSs as a basis for a functional programming language we have to investigate how the PTS-framework can be extended with algebraic data types, case expressions and definitions.

Next to that, we have to construct a type checking algorithm: using PTSs as a basis for a functional programming language requires the ability to type check programs written in the extended PTS language. Unfortunately the type checking problem is not decidable for general PTSs. There exists however an interesting subclass of PTSs for which type checking is decidable. In this thesis we investigate how the type checking algorithm for this class of PTSs can be extended to our language.

Finally, we need to give the *operational semantics* of our extended PTS-language: The operational semantics of a programming language defines in what way a program in the language is executed: in the case of a functional programming language the operational semantics prescribes how terms are evaluated.

Implementation

To be able to experiment with a functional programming language based on PTSs we implement a type checker and an interpreter for our extended PTS language.

Background

We assume that the reader of this thesis understands the syntax and semantics of the programming language Haskell. Furthermore, we assume some basic knowledge of logic, λ-calculus and type theory, in particular the concepts of reduction, substitution, derivability from a set of rules and decidability are assumed to be familiar to the reader.

An introduction to Haskell can be found in [24]. Classical reference guides for the λ-calculus and type-theory are [2] and [3].

Organisation of this thesis

In the next chapter we give a thorough introduction to type systems. We start with the definition of the simply typed λ-calculus, and via the definitions of the different systems of the λ-cube we arrive at the theory of Pure Type Systems. In chapter three we investigate how the theory of PTSs can be extended with algebraic data types, the case construct and definitions. Then, in chapter four we study a type checking algorithm for PTSs and investigate how we can extend the algorithm to deal with our extended language. In chapter five we give a implementation of a type checker and an interpreter for our extended PTS-language. Finally, in chapter six, we conclude and give suggestions for further research.

Acknowledgements

First of all I would like to thank my supervisors Johan Jeuring and Doaitse Swierstra for carefully reading my thesis and their helpful comments. I would like to thank Erik Meijer for introducing me to the idea of using Pure Type Systems in functional programming. Thanks to Henk Barendregt, Herman Geuvers, Wil Dekkers and Jan-Willem Klop for stimulating my enthusiasm about the λ-calculus and type theory during the MRI Masterclass in Mathematical Logic. Thanks to Daan Leijen for giving comments on the grammar and the parser of the language. Last, but not least, I would like to thank my roommates of room B037 for there pleasant company and interesting discussions during the final stage of writing this thesis.

Chapter 2

Type Systems & Functional Programming

2.1 Introduction

The theory of typed lambda calculi lies at the basis of the theory of functional programming. Semantically, a functional programming language can be seen as an extended version of a typed lambda calculus. In that view a functional program comes down to a set of (typable) lambda terms and the evaluation of a program to the reduction of the main term of the program to a normal form.

In this chapter we look at the considerations which play a rôle in choosing a typed lambda calculus (also called a type system) as a basis for a functional programming language. We describe how the characteristics of a type system are reflected in the programming language. We assume that the reader does not have much experience with type systems, so in this chapter we investigate several type systems and their characteristics. We start our journey with the simplest of all type systems, the simply typed lambda calculus, and will end with a quite recent development in the field: the theory of Pure Type Systems.

2.2 The Simply Typed Lambda Calculus

The simply typed lambda calculus, denoted $\lambda\!\!\rightarrow$, lies at the basis of all type systems. It was developed[1] by Curry in 1934[6]. We present a variant of this system, studied by Church in 1940[5]. The difference between the two systems will be explained below.

2.2.1 The definition of $\lambda\!\!\rightarrow$

Definition 2.1 (Types)

Let U be a countable infinite set whose members will be called *type variables*. The set Π of *types* of $\lambda\!\!\rightarrow$ is defined by the following grammar:

$$\Pi ::= U \mid \Pi \rightarrow \Pi$$

[1]or discovered, depending on your view on mathematics

We use $\alpha, \beta, \gamma, \dots$ to denote arbitrary type variables, and $\sigma, \tau, \dots$ to denote arbitrary types. We omit outermost parentheses, and define $\to$ to be right associative.

Definition 2.2 (Terms)
The set $\triangle$ of terms of $\lambda\!\!\to$ is defined by the following grammar:

$$\triangle ::= V | \lambda V : \Pi.\triangle | \triangle\triangle$$

where V is a countable set of term variables. We use $x, y, \dots$ to denote arbitrary term variables, and $M, N, \dots$ to denote arbitrary terms. We adopt the usual conventions for denoting lambda terms.

Definition 2.3 (Reduction)
The relation $\to_\beta$ on E is the smallest relation on $\triangle$ satisfying

$$(\lambda x : A.M)N \to_\beta M[x := N]$$

and closed under the rules

$$\begin{array}{lcll} P \to_\beta P' & \Rightarrow & \forall x \in V, \alpha \in \Pi & : \quad \lambda x : \alpha.P \to_\beta \lambda x : \alpha.P' \\ P \to_\beta P' & \Rightarrow & \forall Z \in \triangle & : \quad PZ \to_\beta P'Z \ \& \ ZP \to_\beta ZP' \end{array}$$

A term M is in *normal form* if there is no term N with $M \to_\beta N$.

$\twoheadrightarrow_\beta$ and $=_\beta$ are the transitive, reflexive closure and the transitive, reflexive, symmetric closure of $\to_\beta$ respectively.

Definition 2.4 (Contexts)
The set C of *contexts* is the set of all sets of pairs of the form

$$x_1 : \tau_1, \dots, x_n : \tau_n$$

with $\tau_1, \dots, \tau_n \in \Pi$, $x_1, \dots, x_n \in V$, and the x_i different. We use Γ, Δ to denote arbitrary contexts.

Definition 2.5 (Typability Relation)
The typability relation $\vdash$ on $C \times \triangle \times \Pi$ is defined by the following typing rules:

$$(\text{Var}) \quad \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$(\to\text{I}) \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau}$$

$$(\to\text{E}) \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

where we require that x does not occur in Γ in the first and second rule.

If $\Gamma \vdash M : \sigma$ we say that M *has type* σ in Γ. We say that M is typable if-and-only-if there exist Γ and σ such that $\Gamma \vdash M : \sigma$.

The set of typable terms is a proper subset of the set of all terms. In this subset restrictions are made regarding which terms may be applied to other terms. Informally type variables denote unspecified sets, and the type $\sigma \to \tau$ denotes the set of functions from σ to τ. If $M : \sigma \to \tau$, then M may only be applied to terms of type σ.

Example 2.1 (Typable terms)
Let σ, τ, ρ be arbitrary types. Then we have:

- $\vdash \lambda x : \sigma.x : \sigma \to \sigma$
- $\vdash \lambda x : \sigma, y : \tau.x : \sigma \to \tau \to \sigma$
- $\vdash \lambda x : \sigma \to \tau \to \rho, y : \sigma \to \tau, z : \sigma.xz(yz) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$

These terms are known as I, K and S.

2.2.2 Type checking and related problems

The following 'problems' are interesting for the use of type systems in functional programming languages. We will explain the importance of the decidability of these problems later on.

Definition 2.6 (Type Checking, Type Inference and Type Correctness)

- The type checking problem is, for a given context Γ, term M and type σ, to answer the question whether $\Gamma \vdash M : \sigma$ holds.
- The type inferencing problem is to find for a given context Γ and term M, a type σ such that $\Gamma \vdash M : \sigma$.
- The type correctness problem is to decide for a given context Γ and term M, whether there exists a type such that $\Gamma \vdash M : \sigma$.

Solving the type inference problem for a certain system implies solving the type correctness problem for that system.

2.2.3 Curry- vs Church Style Typing

As mentioned above, there are two different 'flavours' of the system $\lambda\!\to$. The Curry and the Church style. The difference between the two systems is that in the Church system (the one presented above) the type of a term variable in a lambda abstraction has to be explicitly given. In Curry's system this is not the case.

So in Church's system one writes:

$$\lambda x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha.xz(yz)$$

whereas in Curry's system one writes:

$$\lambda x, y, z.xz(yz)$$

The essential difference between the two approaches is that in the explicit case the unique type of a term can be found easily. For example, in the explicitly typed term above, we already know the types of the variables x, y and z so we can easily derive the type of $xz(yz)$ and thereby the type of the whole term.

In the implicit case types are not unique[2], and the type inference process is considerably harder. For the simply typed lambda calculus there exist algorithms which solve the type inference problem even in the implicit case, but for more complicated type systems this is not the case. And even explicit typing does not guarantee that the type inference problem is decidable for those more complicated systems; below we will discuss explicitly typed type systems for which the type inference problem is nevertheless undecidable.

[2] we have for instance $\vdash \lambda x.x : \alpha \to \alpha$ and $\vdash \lambda x.x : (\alpha \to \alpha) \to (\alpha \to \alpha)$

2.2.4 $\lambda{\to}$ and functional programming

Functional Pseudo Code

To illustrate what kind of programs one could write in a functional programming language based on the type system $\lambda{\to}$, we give some examples of programs which are typable in $\lambda{\to}$, which has been extended with rules for some basic types and functions. To be precise, we extend the set of types with the type `Int`, the set of terms with the set of all integers, and finally we extend the typing rules with the following two rules:

$$(\text{INT}) \qquad \frac{}{n : \texttt{Int}} \quad n \in \mathbf{N}$$

$$(+) \qquad \frac{}{+ : \texttt{Int} \to \texttt{Int} \to \texttt{Int}}$$

To distinguish between pure lambda terms and programs written in a functional programming language, we write the latter code in `this font`.

Abstraction over term variables

The system $\lambda{\to}$ lies at the basis of all type systems for functional programming. The basic characteristic of $\lambda{\to}$ is that it is possible to define functions by lambda abstraction over a term variable.

For instance:

-
  ```
  id : Int -> Int
  id = \x : Int . x
  ```

-
  ```
  twice : (Int -> Int) -> Int -> Int
  twice =  \f:Int -> Int .  \n:Int. f (f n)
  ```

Typable versus untypable terms

In a type system we would like to have the following correspondence:

$$\text{term } P \text{ is untypable} \quad \Leftrightarrow \quad \text{term } P \text{ makes no sense}$$

An example of a term that does not make sense is:

```
+ 3 (\x:Int . x)
```

This term does not make sense because addition is only defined on integers. The reader may check that the term is indeed not typable by the typing rules given above.

Not all type systems conform to the above condition. For example, there are type systems for which there exist terms which make perfect sense to the programmer but which are nevertheless not type correct. Below we will give an example of a meaningful term which is not type correct in $\lambda{\to}$.

But, of course we want more than this condition. It is easy to define a type system in which terms make sense if-and-only-if they are typable: just take the system with a single, meaningless, term that is not typable. This type system is absurd, of course, because we cannot express any programs in it.

So the other important characteristic of type systems concerns expressibility. In type systems we would like to be able to express concepts that arise from the intuition of programmers. In the system $\lambda\rightarrow$ we are able to express the concepts of application of a term on a term, and abstraction of a term over a term. But, next to that, we want to be able to express more complex concepts, such as 'polymorphism' (see section 2.3), dependent types (see section 2.5) or algebraic data types (see section 3.1).

Type Inference & Type Checking

Decidability of type inference and type checking problems is crucial for a type system if we want to use it as the basis for a functional programming language. The reasons for this are the following.

First of all, in a functional programming language we want to be able to filter out meaningless terms at compile time and therefore we need to check whether terms in the program are typable.

If a term is not explicitly typed the compiler uses a type inference algorithm to find out whether the term is typable. If the term is explicitly typed the compiler uses a type checking algorithm to check whether the term is of the given type.

The second reason is that for producing meaningful error messages the compiler needs to be able to infer the types of the subterms of an untypable term.

Implicit vs Explicit Typing

The Church and Curry versions of $\lambda\rightarrow$ can be thought of as representing two different styles of typing in programming languages. Church's system can be compared with programming languages with 'explicit typing', where the programmer has to explicitly write down the types of all variables used in the program.

In programming languages based on Curry's approach the compiler relieves the programmer from this task and derives the types of the variables used in the program itself, which is called 'implicit typing'.

At first hand it seems that implicit typing is the better choice; it is more convenient for the programmer. But, as we have seen above, implicit typing makes type inferencing harder. For some implicitly typed systems the type inference problem is even undecidable. It will be difficult to use such a implicitly typed system as a basis for a functional programming language.

Polymorphic Functions

Unfortunately, it is not possible to define so-called polymorphic functions in $\lambda\rightarrow$. Polymorphic functions are functions that at different calls accept arguments of different types. The prime example of a polymorphic function is the identity function, which takes a term of any type as argument and simply returns that term.

A candidate for the identity function in $\lambda\rightarrow$ would be:

```
id : a -> a
id = \x:a . x
```

The problem is that this function contains a free type variable. For this type variable we can substitute one single type; all occurrences of the function in a program should have the same type substituted for a. Therefore, this function cannot accept arguments of different types at different calls.

A classical example of a term that is intuitively correct, but which is not typable in $\lambda\!\rightarrow$ because of the above reason, is:

```
(id +) (id 3) (id 1)
```

In the next section we deal with this problem by introducing an extension of $\lambda\!\rightarrow$: the second order polymorphic lambda calculus.

2.3 The Second Order Polymorphic Lambda Calculus

The second order polymorphic lambda calculus, denoted $\lambda 2$, was independently developed by the logician Jean-Yves Girard[9] and by the computer scientist John Reynolds[21]. The system $\lambda 2$ gives us the possibility to define *polymorphic* functions.

2.3.1 The definition of $\lambda 2$

Definition 2.7 (Types)
Again, let U be the set of type variables. The set Π of types of $\lambda 2$ is defined by the following grammar:

$$\Pi ::= U \quad | \quad \Pi \rightarrow \Pi \quad | \forall U \Pi$$

The new clause $\forall U \Pi$ generates the types of polymorphic functions.

Definition 2.8 (Terms)
The set $\triangle$ of terms of $\lambda 2$ is defined by the following grammar:

$$\triangle ::= V | \lambda V : \Pi . \triangle | \triangle \triangle | \Lambda U . \triangle | \triangle \Pi$$

There are two new clauses. The first one $\Lambda U.\triangle$, denoting *polymorphic abstraction* generates terms of the form $\Lambda \alpha . M$. The intuitive interpretation of such terms is that the term M (which may contain α at places where types may occur) is a polymorphic function with a type parameter α.

The second new clause, $\triangle\,\Pi$, denoting *type application*, creates terms of the form $(M\tau)$. Such terms are intuitively understood as a call to a generic function M with an actual type parameter τ.

Definition 2.9 (Reduction)

The notion of reduction of $\lambda 2$ is the notion of reduction of $\lambda\!\rightarrow$, extended with *type reduction*.

The relation $\rightarrow_\beta$ is the smallest relation on $\triangle$ satisfying

$$(\lambda x : A.M)N \rightarrow_\beta M[x := N]$$

$$(\Lambda\alpha.M)\tau \rightarrow_\beta M[\alpha := \tau]$$

and closed under the rules

$$\begin{array}{lclcl}
P \rightarrow_\beta P' & \Rightarrow & \forall x \in V, \sigma \in \Pi & : & \lambda x : \sigma.P \rightarrow_\beta \lambda x : \sigma.P' \\
P \rightarrow_\beta P' & \Rightarrow & \forall \alpha \in U & : & \Lambda\alpha.P \rightarrow_\beta \Lambda\alpha.P' \\
P \rightarrow_\beta P' & \Rightarrow & \forall Z \in \triangle & : & PZ \rightarrow_\beta P'Z \\
P \rightarrow_\beta P' & \Rightarrow & \forall Z \in \triangle & : & ZP \rightarrow_\beta ZP' \\
P \rightarrow_\beta P' & \Rightarrow & \forall \tau \in \Pi & : & P\tau \rightarrow_\beta P'\tau
\end{array}$$

Note that in the reduction rule for types, the substitution concerns all type variables and especially the type variables in the λ-abstraction, so we have $(\Lambda\alpha.\lambda x : \alpha.M)\ \mathrm{Int} \rightarrow_\beta \lambda x : \mathrm{Int}.M$.

Definition 2.10 (Typability Relation)
The typability relation $\vdash$ on $C \times \triangle \times \Pi$ is defined as an extension of the rules for $\lambda\!\rightarrow$ (definition 2.5). There are two new rules for the introduction and the elimination of the quantifier:

$$(\forall \mathrm{I}) \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash (\Lambda\alpha.M) : \forall\alpha\sigma} \qquad (\alpha \notin \mathrm{FV}(\Gamma))$$

$$(\forall \mathrm{E}) \qquad \frac{\Gamma \vdash M : \forall\alpha\sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]}$$

The restriction $(\alpha \notin \mathrm{FV}(\Gamma))$ corresponds to the requirement that the type variable which is abstracted from must be a *local* identifier.

Example 2.2 (Typable terms)
Let σ be an arbitrary type. Then we have:

- $\vdash (\Lambda\alpha.\lambda x : \alpha.x) : \forall\alpha.\alpha \rightarrow \alpha$
  This is the polymorphic identity function.
- $\vdash (\Lambda\alpha.\lambda x : \alpha.x)\sigma : \sigma \rightarrow \sigma$
  Applying the polymorphic identity function to a type yields the identity function for terms of this type.
- $\vdash (\Lambda\alpha\beta\gamma.\lambda f : \alpha \rightarrow \beta, g : \beta \rightarrow \gamma, x : \alpha.g(f\ x)) : (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$
  The polymorphic 'composition' function.

2.3.2 Decidability of Type Inference & Type Checking

Theorem 2.1 (Type inference for $\lambda 2$ (Church))
The type inference problem for $\lambda 2$ (Church) is decidable.

Proof: By an easy induction on the structure of the lambda term.

Theorem 2.2 (Type inference for $\lambda 2$ (Curry))
The type inference problem for $\lambda 2$ (Curry) is not decidable.

Proof: See [26]

2.3.3 $\lambda 2$ and functional programming

From now on, we will use `\/` for the pseudo-code of $\forall$ and `/\` for the pseudo-code of Λ.

Polymorphic functions

As mentioned above, the type system $\lambda 2$ allows us to define polymorphic functions. For example, the identity function can be defined by:

```
id :: \/ a . a -> a
id =  /\ a. \x:a . x
```

Notice that the function takes a type as parameter. If we want to apply the function `id` to a term, we first have to instantiate the function to the type of this term. For example `id Int 3`. Using $\lambda 2$, it is possible to define the term which was untypable in $\lambda\!\rightarrow$:

```
(id (Int -> Int -> Int) +) (id Int 3) (id Int 1)
```

Implicit polymorphism and type inference

The kind of polymorphism that is described here is called explicit 'polymorphism': whenever a polymorphic function is used the programmer explicitly has to tell the polymorphic function to which type it should instantiate. It is also possible to define a $\lambda 2$ system in the Curry style, reflecting the idea of 'implicit polymorphism' [22].

Unfortunately, type inference is not decidable for $\lambda 2$ with implicit typing. Therefore $\lambda 2$ in the Curry style is not convenient as a basis for a functional programming language.

There exist several approaches to extend $\lambda\!\rightarrow$ with some kind of polymorphism, while keeping the property that explicit typing is not needed for a decidable type inference problem. All of these systems have in common that some terms which are typable in (the explicitly typed version of) $\lambda 2$ are not typable in the implicitly typed type systems. So, users of languages based on those implicit systems do not have to explicitly type their programs, but in exchange for that they have to take for granted that there exist meaningful programs that will not pass the type checker.

A nice example of a type system where such a trade off is made is the implicitly typed Hindley-Milner system [17]. This system is used as the basis for the programming language Haskell. An example of a term which makes perfect sense but which is not typable in Hindley-Milner is

```
(\f . (f (+)) (f 1) (f 2)) id
```

This fact is illustrated by the printout of a Hugs session in which we ask the interpreter to the evaluate this term:

```
Prelude>  (\f -> f (+) (f 2) (f 3)) id
ERROR: Type error in application
*** Expression     : f (+) (f 2) (f 3)
*** Term           : f 3
*** Type           : b -> a -> c
*** Does not match : a
*** Because        : unification would give infinite type
```

Of course this term can be typed in a functional programming language based on $\lambda 2$. However, due to the explicit typing the term becomes more complicated:

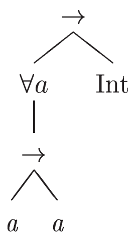
```
(\f: \/ a . a -> a .
f (Int -> Int -> Int) (+) (f Int 1) (f Int 2)) id : Int
```

Example 2.3 (Rank of a type)
The rank of a type is defined as follows: a type has rank $n + 1$ if in the tree representation of the type the maximum of number of left branches of $\rightarrow$ in a single path from the the root to a $\forall$ quantifier is n.

For example:

```
foo :: (forall a. (a -> a)) -> Int
foo f = f (+) (f 3) (f 2)
```

The function `foo` has a rank-2 type:



Unfortunately, the type system of the (rank-2 extended version of) Haskell only supports types up to rank-2, where $\lambda 2$ supports types of arbitrary rank. A real world example of a rank-3 type arises in the mapping of polytypic functions to Haskell programs.

Polytypic functions are functions that are defined by induction on the structure of types. Using such functions it is for instance possible to define equality, pretty-printing and compression functions that work for all types. It is not possible to define polytypic functions in Haskell. It is, however, possible to write a mapping from a polytypic definition and a type to a Haskell function that is the specialisation of the polytypic function to that type. Unfortunately, some specialisations are mapped to functions with a rank-3 type [10, p. 6]. These functions are not typable in Haskell.

An example of a specialisation that yields a rank-3 type is the specialisation of the *map* function to the data type `Sequ`.

```
data Fork a    = Fork a a
data Sequ a    = Empty
               | Zero (Sequ (Fork a))
               | One a (Sequ (Fork a))
```

The data type `Sequ` can be expressed as the fixpoint of a (higher-order) functor. The functor is given by `SequF`, the fixpoint operator is given by `HFix`.

```
data SequF s a = Empty
               | Zero (s (Fork a))
               | One a (s (Fork a))
data HFix h a  = In (h (HFix h) a)

type Sequ'     = HFix SequF
```

If we want to have a map for `Sequ'` we need a map for `SequF` and a map for `HFix`, and then `mapSequ' = mapHFix mapSequF`. The function `mapHFix` has a rank-3 type signature. So a direct mapping of the Generic Haskell specialisation to Haskell is not possible.

We want more...

Using the λ construct we can form terms which depend on other terms, using the Λ construct we can form terms which depend on types. This brings the question forward whether we can we define types which depend on other types?

A natural example of a type depending on another type is $\alpha \to \alpha$ that depends on α. In fact it is tempting to define $f = \lambda\alpha \in \Pi.\alpha \to \alpha$ such that $f\alpha = \alpha \to \alpha$. This is possible in the system Lambda Omega, which we introduce in the next section.

2.4 The System Lambda Omega

The system Lambda Omega, denoted $\lambda\omega$, was introduced by Jean-Yves Girard[9]. The system allows us to abstract types over types.

2.4.1 The definition of $\lambda\omega$

A new feature of $\lambda\omega$ is that types are generated within the system itself and not in the informal meta language. In the previous systems the set of types was generated from a separate type grammar, whereas in $\lambda\omega$ the set of types is defined by the typing rules. We define a new constant $\star$ such that $\sigma : \star$ expresses that $\sigma \in \Pi$ holds. Now, the informal statement (expressed in the type grammar by $\Pi ::= \Pi \to \Pi | \ldots$):

$$\alpha, \beta \in \Pi \Rightarrow (\alpha \to \beta) \in \Pi$$

corresponds to the typing rule:

$$\frac{\Gamma \vdash A : \star \quad \Gamma \vdash B : \star}{\Gamma \vdash (A \to B) : \star}$$

In $\lambda\omega$ we can indeed define a function f such that $f\alpha = \alpha \to \alpha$, this function will be denoted by: $f = \lambda\alpha : \star.\alpha \to \alpha$. The question remains what the type of this function is: The function f takes a type as argument and returns a type, so we choose $f : \star \to \star$. We call $\star \to \star$ a kind. The set of kinds is defined by the following grammar.

$$K ::= \star \,|\, K \to K$$

We introduce a new constant $\Box$ such that $k : \Box$ corresponds to $k \in K$. The set of kinds is, just as set of types, generated by the typing rules of $\lambda\omega$.

Definition 2.11 (Expressions of $\lambda\omega$)

The set *expressions* E of $\lambda\omega$ is formed by the following grammar:

$$E ::= V \mid \star \mid \Box \mid EE \mid \lambda V : E.E \mid E \to E \mid \Lambda E.E | \forall E : E.E$$

Note that instead of defining three different sets of terms, types and kinds, we only define the set of expressions. The typing rules determine which of the expressions are terms, types or kinds.

Actually, we have to be more precise. We say that an expression E has:

- sort kind if-and-only-if $E : \Box$.
- sort type if-and-only-if there exist an expression K of sort kind such that $E : K$.
- sort term if-and-only-if there exists an expression T of sort type such that $E : T$.

Often, we say that an expression *is* a kind, type or term, instead of saying that an expression *has sort* kind, type or term. This can lead to ambiguity. Because by saying that E is a type we sometimes mean that $E : \star$, which is stronger than saying that E has sort type. For instance, $f = \lambda\alpha : \star.\alpha \to \alpha : \star \to \star$ has sort type because $\star \to \star : \Box$, but we don't have $f : \star$. To prevent this kind of ambiguity, we say that E has:

- *kind* type when we mean that $E : \star$
- *sort* type when mean there exists a K such that $E : K$ and $K : \Box$

and we try to avoid saying that E is a type.

Another source of confusion is the use of the phrase *A has type B*. This phrase does not imply that B has kind or sort type. (For instance we often say that $\star$ has type $\Box$.) The only thing that we express by saying that A has type B (in context Γ) is that $\Gamma \vdash A : B$

The question remains what we mean by types depending on types. Do we mean that expressions of kind type can depend on expressions of kind type, or do we mean that expressions of sort type can depend on expressions of sort type? The latter is the case. This can be explained by looking at the abstraction rule for types over types, which is described on the next page. To abstract $b : B$ over $x : A$ we need that $(A \to B)$ has type $\Box$. So we need the third type formation rule, which demands that $A, B : \Box$. This means that $x : A : \Box$ and $b : B : \Box$, so x and b are both of *sort* type.

Definition 2.12 (Reduction)
The reduction relation $\to_\beta$ for $\lambda\omega$ is the same as the reduction relation for $\lambda 2$.

Definition 2.13 (Contexts)
The set C of *contexts* is the set of all lists of the form

$$[x_1 : A_1, \ldots, x_n : A_n]$$

with $A_1, \ldots, A_n \in E$ and $x_1, \ldots, x_n \in V$. We use [] to denote the empty list, and if $X = [x_1 : A_1, \ldots, x_n : A_n]$ and $Y = [y_1 : B_1, \ldots, y_n : B_n]$, then $X, x : A = [x_1 : A_1, \ldots, x_n : A_n, x : A]$ and $X +\!\!+ Y = [x_1 : A_1, \ldots, x_n : A_n, y_1 : B_1, \ldots, y_n : B_n]$.

Contexts of $\lambda\omega$ are lists instead of sets. The reason is that in $\lambda\omega$ the order in which assumptions are made is important. The (start) rule dictates that before we can assume $x : \alpha$, we have make an assumption about α. The abstraction rules dictates that the assumptions should be discharged in the reverse order as in which they are made. Changing the order of the assumptions along the way yields strange results: For instance using the (start) and (axiom) rule we can derive $\alpha : \star, x : \alpha \vdash x : \alpha$. Changing the order would lead to $x : \alpha, \alpha : \star \vdash x : \alpha$. Then we could derive $x : \alpha \vdash (\Lambda\alpha : \star.x) : (\forall\alpha : \star.\alpha)$ and thus $x : \alpha \vdash (\Lambda\alpha : \star.x)$ Int : Int. Using the fact that $\lambda\omega$ has the *subject reduction property* (that is $\Gamma \vdash A : \tau$ and $A \rightarrow_\beta A'$ implies $\Gamma \vdash A' : \tau$) we would arrive at $x : \alpha \vdash x : \text{Int}$ which is clearly wrong.

Definition 2.14 (Typability Relation)
The typability relation $\vdash$ on $C \times E \times E$ is defined by the typing rules in figure 2.4.1. In the rules s ranges over $\star, \Box$.

First of all, the axiom rule states that the constant $\star$ has type $\Box$. This defines $\star$ to be a kind, all other kinds are generated by the kind formation rule. The start rule states that if the type A is well formed, then we can derive $x : A$ from any context which ends with $x : A$. The weakening rule says that we can throw away the irrelevant binding $x : C$ as long as the expression C is well formed. The next two rules describe how types can be formed. The first allows us to define function types. The second rule can be used to define polymorphic types. The kind formation rule describes how kinds can be formed. The three application rules describe how expressions can be applied to other expressions. The abstraction rules tell us which abstractions may be formed. The final rule is the conversion rule. It tells us that if we can deduce that a has type A, and we can deduce that A is β-equal to the well-formed expression B, that we may deduce that a has type B. This rule is necessary because in $\lambda\omega$ reduction is possible on the level of types, so types, like terms, do not have to be in normal form. For instance, suppose that $\text{f} = \lambda\alpha : \star.\alpha$ and $\text{id} = \Lambda\alpha : *.\lambda x : \alpha.x$, then we have $\vdash \text{id (f Int)} : \text{(f Int)} \rightarrow \text{(f Int)}$. The conversion rule allows us to make the necessary reduction to deduce $\vdash \text{id (f Int)} : \text{Int} \rightarrow \text{Int}$.

Example 2.4 (Typable terms)

- $\vdash (\lambda\alpha : \star.\alpha \rightarrow \alpha) : \star \rightarrow \star$
  An identity function for types of kind $\star$.
- $\vdash (\lambda\alpha : \star \rightarrow \star.\alpha) : (\star \rightarrow \star) \rightarrow \star \rightarrow \star$
  An identity function for types of kind $\star \rightarrow \star$
- define $\text{compose} = \lambda\alpha : \star \rightarrow \star.\beta : \star \rightarrow \star.\gamma : \star.\alpha\ (\beta\ \gamma)$, then:
  - $\vdash \text{compose} : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$
  - $\text{List} : \star \rightarrow \star, \text{Maybe} : \star \rightarrow \star \vdash \text{compose List Maybe Int} : \star$

2.4.2 Decidability of Type Inference

Theorem 2.3 (Type inference for $\lambda\omega$ (Church))
The type inference and type checking problems for $\lambda\omega$ (Church) are decidable.

Proof: Direct consequence of theorem 2.7.

Theorem 2.4 (Type inference for $\lambda\omega$ (Curry))
The type inference problem for $\lambda\omega$ (Curry) is not decidable.

Proof: Direct consequence of theorem 2.2.

(axiom)	$\dfrac{}{[] \vdash \star : \square}$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$
(weak)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$
(type formation 1)	$\dfrac{\Gamma \vdash A : \star \quad \Gamma \vdash B : \star}{\Gamma \vdash (A \to B) : \star}$
(type formation 2)	$\dfrac{\Gamma \vdash A : \square \quad \Gamma, x : A \vdash B : \star}{\Gamma \vdash (\forall x : A.B) : \star}$
(kind formation)	$\dfrac{\Gamma \vdash A : \square \quad \Gamma \vdash B : \square}{\Gamma \vdash (A \to B) : \square}$
(application of terms on terms)	$\dfrac{\Gamma \vdash F : (A \to B) \quad \Gamma \vdash a : A \quad \Gamma \vdash A : \star}{\Gamma \vdash Fa : B}$
(application of types on types)	$\dfrac{\Gamma \vdash F : (A \to B) \quad \Gamma \vdash a : A \quad \Gamma \vdash A : \square}{\Gamma \vdash Fa : B}$
(application of terms on types)	$\dfrac{\Gamma \vdash F : (\forall x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$
(abstraction of terms over terms)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (A \to B) : \star}{\Gamma \vdash (\lambda x : A.b) : (A \to B)}$
(abstraction of types over types)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (A \to B) : \square}{\Gamma \vdash (\lambda x : A.b) : (A \to B)}$
(abstraction of types over terms)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\forall x : A.B) : \star}{\Gamma \vdash (\Lambda x : A.b) : (\forall x : A.B)}$
(conversion rule)	$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash B : s \quad A =_\beta B}{\Gamma \vdash a : B}$

Figure 2.1: Typing rules of $\lambda\omega$

2.4.3 $\lambda\omega$ and functional programming

$\lambda\omega$ enables us to define types which depend on other types. For example:

```
id_type : * -> *
id_type : \a:*. a -> a

id : \/a:* . id_type a
id = /\a:* . \x : a . x
```

In the next chapter, we explain how to define so called *algebraic data types* in type systems. It turns out that we need types depending on types to be able to describe these data types.

2.5 The Calculus of Constructions

$\lambda\omega$ enables us to define terms that depend on terms, terms that depend on types and types that depend on types. The only thing missing is types that depend on terms.

The standard example of a useful type that depends on a term is the type of *generalised zip*. The Haskell prelude provides the functions `zip`, `zip2`, …, `zip4`. When a programmer needs a function that zips, say 27 lists together (s)he has to write his/her own `zip27` function. Of course, this is ugly. It would be nice if we could *abstract* over the number of lists that need to be zipped. Then we need a function that takes an integer as an argument and yields the corresponding zip function, such that:

```
gen_zip 2  : \/a1,a2 . [a1] -> [a2] -> [(a1,a2)]
gen_zip 3  : \/a1,a2,a3 . [a1] -> [a2] -> [a3] -> [(a1,a2,a3)]
gen_zip 27 : \/a1,...,a27 . [a1] ->...-> [a27] -> [(a1,...,a27)]
```

So, the type of `gen_zip` depends on a term (here an integer). The *Calculus of Constructions* introduced by Coquand and Huet[23], denoted λC, gives us the flexibility to define such types.

2.5.1 Reflection on abstraction

We have seen three different dependencies above:

- terms depending on terms.
  In $\lambda\!\rightarrow$ we can define terms depending on terms. Given a term M, we may form the term $\lambda x : \sigma.M$ which expects a term as argument. In other words $\lambda x : \sigma.M$ is a term that depends on a term. An example of a term depending on a term is: $(\lambda x : \sigma.x) : \sigma \rightarrow \sigma$.

- terms depending on types.
  In $\lambda 2$ we can define terms depending on types. Given a term M, we may form the term $\Lambda\alpha.M$ which expects a type as argument. In other words $\Lambda\alpha.M$ is a term that depends on a type. An example of term depending on a type is: $(\Lambda\alpha.\lambda x : \alpha.x) : \forall\alpha.\alpha \rightarrow \alpha$

- types depending on types.
  In $\lambda\omega$ we can define types depending on types. Given a type σ, we may form the type $\lambda\alpha : \kappa.\sigma$ which expects a type of kind κ as argument. In other words $\lambda\alpha : \kappa.\sigma$ is a type that depends on a type. An example of type depending on a type is: $(\lambda\alpha : \star.\alpha \to \alpha) : \star \to \star$.

The typing rules for $\lambda\omega$ above provide different abstraction and application rules for each of the three dependencies above. The system λC allows one extra dependency: types depending on terms. Although λC can be defined by a set of extra abstraction and application rules for types depending on terms, we choose to give a more concise set of rules, in which there are only two rules which provide all the forms of abstraction and application. In that way we can easily generalise λC to the so called λ-cube, of which all the systems discussed above are components.

We introduce a new syntax, the dependent product Π:

$$\Pi x : A.B$$

means the type of functions from values of type A to values of type B, in which the type B may depend on the value of the argument x.

We can use the dependent product to describe all four of the above dependencies.

First, lets look at the type of a term M that depends on a term. Lets assume that M applied to a term of type A, yields a term of type B. (in $\lambda\!\to$ we we would say $M : A \to B$). Then M is nothing else than a function from values of type A to values of type B, in which the type B does not depend on the argument! So, we can say $M : \Pi x : A.B$, where x does not occur free in B. Or we can can say $M : \Pi\_ : A.B$, where $\_$ denotes an anonymous variable. In both cases we have $A, B : \star$.

The same applies for types depending on types. For example, say $f : \star \to \star$. Then f is nothing else than a function from values of type $\star$ to values of type $\star$. So, we can say $f : \Pi\_ : \star.\star$. In general we can type types that depend on types with $\Pi x : A.B$ with $A, B : \Box$.

So we will use the following convention:

$$A \to B \text{ is an abbreviation for } \Pi\_ : A.B$$

Now, lets look at the type of a term depending on a type. For example, $M : \forall\alpha : \star.B$, with $B : \star$. Then M is a function from values of type $\star$ to values of type B, in this case B may depend on α. So $M : \Pi\alpha : \star.B$, In general we can type terms depending on types with $\Pi x : A.B$ with $A : \Box$ and $B : \star$.

We define:

$$\forall\alpha : A.B \text{ is an abbreviation for } \Pi\alpha : A.B$$

Finally, lets look at the type of types depending on terms. These are functions from terms to types in which the result type may depend on the value of the term argument. Such functions have type $\Pi x : A.B$ where $A : \star$ and $B : \Box$.

For all these types we have one single type formation rule:

$$\text{(PI)} \qquad \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t} \qquad (s,t) \in \{\star, \Box\} \times \{\star, \Box\}$$

Depending on the choice for s and t we can type terms depending on terms, terms depending on types, types depending on types and types depending on terms. We have the following correspondence:

$(\star,\star)$	terms depending on terms
$(\Box,\star)$	terms depending on types
$(\Box,\Box)$	types depending on types
$(\star,\Box)$	types depending on terms

All four abstractions are dealt with in one rule:

$$\text{(LAM)} \qquad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \qquad t \in \{\star, \Box\}$$

There is also one single application rule:

$$\text{(APP)} \qquad \frac{\Gamma \vdash f : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$$

Applying the abbreviations given above, the reader can check that these rules correspond to the rules of $\lambda\omega$ for the first three dependencies.

2.5.2 The definition of λC

Now, we present the formal definition of the type system λC.

Definition 2.15 (Expressions)
The set *expressions* E of λC is formed by the following grammar:

$$E ::= V \mid \star \mid \Box \mid EE \mid \lambda V : E.E \mid \Pi V : E.E$$

Definition 2.16 (Reduction)
The relation $\to_\beta$ is the smallest relation on E satisfying

$$(\lambda x : A.M)N \to_\beta M[x := N]$$

and closed under the rules

$$\begin{array}{lcll}
P \to_\beta P' & \Rightarrow & \forall x \in V, A \in E & : \quad \lambda x : A.P \to_\beta \lambda x : A.P' \\
P \to_\beta P' & \Rightarrow & \forall x \in V, A \in E & : \quad \Pi x : A.P \to_\beta \Pi x : A.P' \\
P \to_\beta P' & \Rightarrow & \forall Z \in E & : \quad PZ \to_\beta P'Z \\
P \to_\beta P' & \Rightarrow & \forall Z \in E & : \quad ZP \to_\beta ZP'
\end{array}$$

Definition 2.17 (Contexts)
The set C of contexts of λC is defined in the same way as the set of contexts of $\lambda\omega$.

Definition 2.18 (Typability Relation)
The typability relation $\vdash$ on $C \times E \times E$ is defined by the following typing rules:

(axiom) $$\frac{}{[] \vdash \star : \Box}$$

(start) $$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

(weak) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$$

(PI) $$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t} \quad (s,t) \in \{\star, \Box\} \times \{\star, \Box\}$$

(LAM) $$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \quad t \in \{\star, \Box\}$$

(APP) $$\frac{\Gamma \vdash f : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$$

(CONV) $$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : S \quad A =_\beta B}{\Gamma \vdash a : B}$$

Example 2.5 (Typable terms)

It is hard to give examples of terms with dependent types that are meaningful in a functional programming context without using algebraic data types and case expressions (which will be introduced in the next chapter). Therefore we will give some examples of typable terms that have no direct connection with functional programming, but nevertheless illustrate the use of typing rules.

- $\alpha : \star \vdash \alpha \to \star : \Box$
  Types may depend on terms!
- $\vdash (\lambda x : \alpha.\alpha) : \alpha \to \star$
  A simple term with a type that depends on a term.
- $d : \text{Int} \to \star \vdash d\ 3 : \star$
  Applying a ‘dependent type’ to a term yields a type.

2.5.3 Decidability of Type Inference & Type Checking

Theorem 2.5 (Type inference for λC (Church))

The type inference and type checking problems for λC (Church) are decidable.

Proof: Direct consequence of theorem 2.7.

Theorem 2.6 (Type inference for λC (Curry))

The type inference problem for λC (Curry) is not decidable.

Proof: Direct consequence of theorem 2.2.

2.5.4 λC and functional programming

λC allows us to define types that depend on terms. In the introduction of this section we have given an example that shows how these so-called 'dependent types' can be useful. However, to be able to give meaningful terms with 'dependent types' we need to extend our language with algebraic data types and case expressions. Therefore, we postpone examples of dependent types to the next chapter. Another source of examples of the use of dependent types in functional programming is the documentation of Cayenne[1].

2.6 The Lambda Cube

In this section we introduce Barendregt's λ-*cube*[3]. The λ-cube can be seen as a generalisation of eight type systems, among these are the systems $\lambda\!\rightarrow$, $\lambda 2$, $\lambda\omega$ and λC which were introduced in the previous sections.

Most parts of the definition of the λ-cube have already been given in the previous section. In our description of the type system λC we already had the λ-cube in mind. Therefore, the definition of the λ-cube is just a minor modification of the definition of the system λC.

2.6.1 The definition of the λ-cube

Definition 2.19 (Expressions and contexts)

The sets of expressions and contexts of the λ-cube are the same as the sets of expressions and contexts of the system λC.

Definition 2.20 (Reduction)

The reduction relation $\rightarrow_\beta$ for the λ-cube is the same as the reduction relation for λC .

Definition 2.21 (Typability Relation)

For any set R such that $\{(\star,\star)\} \subseteq R \subseteq \{\star,\Box\} \times \{\star,\Box\}$ the relation $\vdash$ on $C \times E \times E$ is defined by the typing rules of λC except for the PI rule which becomes:

$$\text{(PI)} \qquad \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t} \qquad (s,t) \in R$$

By choosing appropriate subsets of $\{\star,\Box\} \times \{\star,\Box\}$ for R the λ-cube instantiates to eight type systems. See figure 2.2. For instance the set of rules of the system λP is $\{(\star,\star),(\star,\Box)\}$.

2.6.2 Decidability of Type Inference

Theorem 2.7 (Type inference for the λ-cube)

The type inference problems for all the systems of the λ-cube (in the Church style) are decidable.

Proof: Direct consequence of theorem 2.7.3.

system	names and references	Set of Rules (R)			
$\lambda\!\rightarrow$	simply typed λ-calculus [5] [6]	$(\star,\star)$			
$\lambda 2$	second order λ-calculus, system F [9] [21]	$(\star,\star)$	$(\Box,\star)$		
$\lambda\underline{\omega}$	[8]	$(\star,\star)$		$(\Box,\Box)$	
$\lambda\omega$	lambda omega, system Fω[9]	$(\star,\star)$	$(\Box,\star)$	$(\Box,\Box)$	
λP	[7]	$(\star,\star)$			$(\star,\Box)$
λP2	[15]	$(\star,\star)$	$(\Box,\star)$		$(\star,\Box)$
$\lambda P\underline{\omega}$		$(\star,\star)$		$(\Box,\Box)$	$(\star,\Box)$
λC	calculus of constructions [23]	$(\star,\star)$	$(\Box,\star)$	$(\Box,\Box)$	$(\star,\Box)$

Figure 2.2: The systems of the λ-cube

2.6.3 The λ-cube and functional programming

The λ-cube framework generalises a rich set of type systems, supporting polymorphism, functions on types, and dependent types. By simply selecting or discarding rules one can force the framework to support or not support each of these features. This flexibility and ability to parameterise makes the λ-cube a good choice for a basis for a functional programming language.

Furthermore, the λ-cube provides a single syntax for terms, types and kinds, this makes it possible to use a single data type in implementations to represent all three levels, and to use a single set of utility functions (like parsing, pretty-printing, substitution functions) that works on all levels. This leads to considerable code efficiency when writing tools (like compilers or interpreters) for languages based on the λ-cube.

Unfortunately, despite its flexibility, there are still things that are not possible in the λ-cube framework, as we will see in the next section.

2.7 Pure Type Systems

Using the systems in the λ-cube we can define a number of different identity functions. Using $\lambda\!\rightarrow$ we can define the monomorphic identity function on terms (of type α) by $(\lambda x : \alpha.x) : \alpha \rightarrow \alpha$. Using $\lambda 2$ we can define the polymorphic identity function on terms by $(\Lambda\alpha.\lambda x : \alpha.x) : \forall\alpha.\alpha \rightarrow \alpha$. Using $\lambda\omega$ we can define the monomorphic identity function on types (of kind κ) by $(\lambda\alpha : \kappa.\alpha) : \kappa \rightarrow \kappa$. But this is where hierarchy stops: it is not possible to define a 'polymorphic identity function on types'. That is: it is not possible to define a function which takes a kind κ as argument and yields the identity function on types of kind κ. It is tempting to define such a function by: $(\lambda\kappa : \Box.\lambda\alpha : \kappa.\alpha) : \Pi\kappa : \Box.\kappa \rightarrow \kappa$. Although the expression is syntacticly valid, it is not possible to type it by the type system of the λ-cube. To be able to type the expression by using the (LAM) rule, we need $\Pi\kappa : \Box.\kappa \rightarrow \kappa$ to be typable. The premises of the (PI) rule state (twice) that we need to assign a type to $\Box$. Because there is not a rule which assigns a type to $\Box$ the candidate function cannot be typed. If we would however extend the sorts with an extra sort, say $\Box'$, and add an extra axiom rule $[] \vdash \Box : \Box'$ we would be able to define a 'polymorphic identity function on types'.

The example above illustrates that the λ-cube is not general enough for all purposes: the type system described above takes us outside the scope of the λ-cube. Fortunately, there is well studied generalisation of the λ-cube in which the described system fits. This generalisation is called the the theory of PTSs. The main differences between the λ-cube and PTSs are that

- in PTSs one can choose the set of sorts S freely, where in the λ-cube the set of sorts is fixed to $\{\star, \Box\}$.
- a relation $A \subseteq S^2$ is defined as the set of axioms, instead of the single axiom $\star : \Box$.
- the product rule is generalised in the sense that products need not to have the same type as their range. That is, $\Pi x : A.B$ does not necessarily have the same sort as B. This is reflected in the fact that rules have three components in PTSs instead of two in the λ-cube.
- the set of rules R can be any subset of S^3 instead of the specific rules from figure 2.2.

2.7.1 The definition of PTSs

Definition 2.22 (Pure Type System)
A Pure Type System is a triple (S, A, R) where

- S is a set of *sorts*,
- $A \subseteq S^2$ is a set of *axioms*,
- $R \subseteq S^3$ is a set of *rules*.

Sometimes we denote rules of the form $(s_1, s_2, s_2) \in S^3$ by $(s_1, s_2) \in S^2$.

Definition 2.23 (Expressions)
The set *expressions* E of a PTS is formed by the following grammar:

$$E ::= V \mid S \mid EE \mid \lambda V : E.E \mid \Pi V : E.E$$

where V is a countable set of variables.

Definition 2.24 (Reduction)
The reduction of expressions of a PTS is defined is the same way as for λC.

Definition 2.25 (Contexts)
The set of contexts of a PTS is defined is the same way as for λC.

Definition 2.26 (Typability Relation of a PTS)
The typability relation $\vdash$ on $C \times E \times E$ for PTSs is defined by the following typing rules:

(axiom)	$\dfrac{}{[] \vdash s_1 : s_2}$	$(s_1, s_2) \in A$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \mathrm{dom}(\Gamma)$
(weak)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$	$x \notin \mathrm{dom}(\Gamma)$
(PI)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3}$	$(s_1, s_2, s_3) \in R$
(LAM)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$	
(APP)	$\dfrac{\Gamma \vdash f : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$	
(CONV)	$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash B : S \quad A =_\beta B}{\Gamma \vdash a : B}$	

2.7.2 Examples of PTSs

Example 2.6 (The λ-cube as a set of PTSs)
The λ-cube consists of eight PTSs, where:

1. $S = \{\star, \Box\}$
2. $A = \{(\star, \Box)\}$
3. $\{(\star, \star)\} \subseteq R \subseteq \{(\star, \star), (\star, \Box), (\Box, \star), (\Box, \Box)\}$

Example 2.7 (An extension of $\lambda\omega$)
Above we described a system in which the 'polymorphic identity function on types' can be defined. This system can be defined as a PTS by:

1. $S = \{\star, \Box, \Box'\}$
2. $A = \{(\star, \Box), (\Box, \Box')\}$
3. $R = \{(\star, \star), (\Box, \star), (\Box, \Box), (\Box', \Box')\}$

In this PTS we have $\vdash (\lambda\kappa : \Box.\lambda\alpha : \kappa.\alpha) : \Pi\kappa : \Box.\kappa \to \kappa$.

Example 2.8 (A predicative variant of $\lambda\omega$)
In $\lambda\omega$ objects can be abstracted from universally-quantified types. This has the (theoretical) drawback that it is hard to define a model for the calculus. Using PTSs this problem can be 'fixed' by forbidding abstractions from universally-quantified types. See [12].

In stead of just the sorts $\star$ and $\Box$ we define four sorts:

$\star$	is the kind of 'monotypes', with super-kind $\Box$,
$\star\star$	is the kind of 'polytypes', with super-kind $\Box\Box$.

The systems comes with the following rules:

1. $S = \{\star, \star\star, \Box, \Box\Box\}$
2. $A = \{(\star, \Box), (\star\star, \Box\Box)\}$
3. $R = \{(\star,\star,\star), (\star,\star\star,\star\star), (\star\star,\star,\star\star), (\star\star,\star\star,\star\star), (\Box,\star,\star\star), (\Box,\star\star,\star\star), (\Box,\Box,\Box)\}$

The first four rules determine how mono- and polytyped terms interact. If one abstracts a term from a term, and at least one of the terms is polytyped, then the resulting term will be a polytyped term too. The following two rules, $(\Box, \star, \star\star)$ and $(\Box, \star\star, \star\star)$, say that it is permitted to abstract a monotype from either a monotyped or a polytyped term, in both case the resulting term is polytyped. Finally, the last rule, $(\Box, \Box, \Box)$, states that it is permitted to abstract monotypes from monotypes. The import thing is that the rules prevent that one can abstract polytypes from anything; there is no rule of the form $(\Box\Box, s_2, s_3)$.

Note that in the third and fifth rule the types of s_2 and s_3 differ, that means that the type of object that is abstracted over, and the result type are not the same. So, this example illustrates the usefulness of the three-placed rules (as opposed to the two-placed ones in the λ-cube) of PTSs.

Assuming that $\text{Int} : \star$ on can derive the following judgements in this PTS:

- $\vdash \text{Int} : \star$
  Int is a monotype.
- $\vdash \text{Int} \to \text{Int} : \star$
  Abstracting a monotyped term variable from a monotyped term yields term with a monotype.
- $\vdash \forall\alpha : \star.\alpha \to \text{Int} : \star\star$
  Abstracting a monotype from a monotyped term (of type Int) yields a term with a polytype.
- $\vdash (\forall\alpha : \star.\alpha \to \text{Int}) \to \text{Int} : \star\star$
  Abstracting a polytyped term-variable (of type $(\forall\alpha : \star.\alpha \to \text{Int})$) from a monotyped term (of type Int) yields a term with a polytype.

2.7.3 Decidability of Type Inference & Type Checking

Type checking and inference are not decidable for every PTS. Fortunately, they are decidable for an important class of PTSs.

Theorem 2.8 (Decidability of type checking and type inference for PTSs) Let (S, A, R) be a PTS. If S is finite then type checking and type inference are decidable.

Proof: See [25].

2.7.4 PTSs and functional programming

The framework of PTSs adds another degree of flexibility to the λ-cube by parameterising the set of sorts and the rules and axioms determining the possible interactions between those sorts. It seems that up till now there is no urgent need to introduce other sorts or rules than the ones available in the λ-cube.

Nevertheless, the examples above illustrate the power of the framework of PTSs, and it would not be surprising that soon useful features of functional programming languages are developed that need the extra flexibility of PTSs.

Therefore, we think it is good idea to use PTSs as as basis for a functional programming language, so that we not need to redevelop our theory, and to develop new tools, once we discover that we need to go beyond the λ-cube.

Chapter 3

Towards a programming language

Although type systems lie at the basis of functional programming languages, a type system -in the mathematical sense- alone is not enough to describe a functional programming language. First of all, functional programming languages provide features, such as algebraic data types, case expressions and definitions, whose description lies outside the scope of type systems. To describe these features we need to extend the type system definitions of the previous chapter. Furthermore, we need to give the *operational semantics* of our language, that is we have to describe how expressions of our language are evaluated.

In this chapter we describe how we extend the PTS framework to deal with algebraic data types, case expressions and definitions. Next to that we will define the operational semantics of our language.

3.1 Algebraic Data Types

Functional programming languages provide means for users to define their own data structures, called algebraic data types. In this section we will explain how we can extend the PTS framework to deal with algebraic data types.

3.1.1 Algebraic Data Types in Haskell

In this subsection we give several examples of Haskell definitions of algebraic data types.

```
data Bool     = True | False

data List a   = Nil | Cons a (List a)

data Tree a f = Leaf a | Branch (f (Tree a f))
```

The first line of code above defines `Bool` to be a new expression of sort type, in the λ-cube formalism we would say `Bool:*`. `True` and `False` are so-called *data constructors*, the only way to create expressions of type `Bool` is by using these constructors. We have `True:Bool` and `False:Bool`.

The next line of code defines `List` to be a new type constructor, if `a` is a type then `List a` is a type as well. In the λ-cube formalism we would say `List:* -> *`. The `List` type constructor comes with two data constructors. The first is `Nil` which stands for the empty list, for every type `a` we have `Nil:List a`. The second constructor is `Cons` which constructs a new list from an element and a list, for every type `a` we have `Cons:a -> List a -> List a`.

Finally, the last line of code defines another, more complex, type constructor; the type constructor of *generalised trees*. We have: `Tree : * -> (* -> *) -> *` For every two expressions `a` and `f` such that `a:*` and `f:* -> *` we have `Leaf : a -> Tree a f` and `Branch : f (Tree a f) -> Tree a f`. As an example: `Branch [Leaf True, Leaf False] : Tree Bool List`. (Where `[ , ]` is the standard Haskell sugar for lists).

Definition 3.1 (Definition of ADTs in Haskell)
In Haskell the general form of an algebraic type definition is (see [11]):

$$\begin{array}{lllllll}
\texttt{data}\ tc\ tca_1 \ldots tca_{\#tca} & = & dc_1 & dcat_{1,1} & \ldots & dcat_{1,\#dca_1} \\
 & | & dc_2 & dcat_{2,1} & \ldots & dcat_{2,\#dca_2} \\
 & \vdots & \vdots & \vdots & & \vdots \\
 & | & dc_{\#dc} & dca_{\#dc,1} & \ldots & dcat_{\#dc,\#dcat_{\#dc}}
\end{array}$$

We call tc a *type constructor* and $tca_1 \ldots tca_{\#tca}$ the arguments of the type constructor. We call $dc_1 \ldots dc_{\#dc}$ the *data constructors* of tc, and the $dcat_{jk}$ the types of the arguments of the data constructors. We use $\#tca$ to denote the number of type constructor arguments, $\#dc_j$ to denote the number of data constructor arguments of data constructor j, etc.

Each tca_i is a variable of sort type, each $dcat_{jk}$ is an expression of kind type (that is $dca_{jk} : \star$) which may contain tc and $tca_1 \ldots tca_{\#tca}$ as free variables.

In the algebraic data type definition the types of the tca_l's are not specified; the compiler or interpreter is supposed to derive them. If we assume that $tca_l : tcat_l$, then the type and data constructors have the following types.

$$tc : tcat_1, \rightarrow ldots \rightarrow tcat_{\#tca} \rightarrow \star$$

and for all $1 \leq j \leq \#dc$:

$$\begin{array}{ll}
dc_j : & \forall tca_1 : tcat_1, \ldots, tca_{\#tca} : tcat_{\#tca}. \\
 & dca_{j,1} \rightarrow \ \ldots \rightarrow dca_{j,\#dca_j} \rightarrow \\
 & (tc\ tca_1 \ldots tca_{\#tca})
\end{array}$$

3.1.2 Algebraic Date Types in PTSs

A simple way to introduce ADTs in a PTS is to treat newly defined type- and data constructors as typed variables. In that way a PTS with ADTs is a tuple which consists of a PTS and a list of typed variables. The link between the two is that the set of typed variables is always assumed to be part of the context which is used in the typability relation.

For instance, if we want to introduce the Bool type in a PTS, we want the following variables in the context:

$$\text{Bool} : \star,\ \text{True} : \text{Bool},\ \text{False} : \text{Bool}$$

In the Haskell examples above, the type definitions are implicitly typed, the compiler needs to derive the type of the type- and data constructors. In our PTS framework the type definitions should be explicitly typed. This means that in an algebraic type definition the types of the type- and data constructors should be explicitly provided.

For instance, if we want to introduce the List type we need to specify the types of List, Nil and Cons. So, to introduce the List type in a PTS, we have to add the following variables to the context:

$$\text{List} : \star \to \star,\ \text{Nil} : \forall a : \star.\text{List}\ a,\ \text{Cons} : \forall a : \star.a \to \text{List}\ a \to \text{List}\ a$$

Furthermore, when using data constructors to form expressions we have to explicitly provide the types at which the data constructors are ‘instantiated’. For instance, the empty list of integers is constructed by Nil Int, instead of just Nil as in Haskell.

As a last example, to introduce the Tree data type in a PTS, we have to add the following variables to the context:

$$\text{Tree} : \star \to (\star \to \star) \to \star, \text{Leaf} : \forall a : \star, f : \star \to \star.a \to \text{Tree}\ a\ f$$

$$\text{Branch} : \forall a : \star, f : \star \to \star.f\ (\text{Tree}\ a\ f) \to \text{Tree}\ a\ f$$

The definition of a PTS with ADTs

Defining algebraic data type in PTSs is a delicate matter. As stated above, we have chosen to extend a PTS with ADTs by adding a set of typed variables to the context. The question is: which sets of typed variables should be allowed? A conservative choice would be to allow only sets of typed variables that meet the definition of Haskell ADTs (see definition 3.1). We feel this choice would be too conservative: Haskell (without extensions) only allows independent data constructor arguments. The following data type definition is therefore not allowed: a type constructor $\text{E} : \star$ with a single data constructor $\text{EC} : \forall a : \star.a \to (a \to \text{Int}) \to \text{E}$. In this definition the type of the second and the third argument of the data constructor depend on its first argument. This kind of constructions are essential for the definition of so called *existential datatypes*, (see example 3.3). Therefore, we have chosen for a more liberal definition than the Haskell definition. Our definition is based on the following requirements, that we feel should be met before we call a set of typed variables a data type:

- A type constructor, say tc, has a number of arguments, called type constructor arguments. A type constructor applied to the right number of arguments yields an expression of kind type.
- A data constructor has as first arguments the arguments of its type constructor, next to that a data constructor has a number of data constructor arguments. A data constructor applied to the right number of type- (say $tca_1 \ldots tca_n$) and data constructor arguments yields an expression of type $tc\ tca_1 \ldots tca_n$.

Assuming that the type constructor is a variable named tc, which takes $\#tca$ type arguments and is equipped with $\#dc$ data constructors named $dc_1, \ldots, dc_{\#dc}$ which each take $\#dca_j$ data constructor arguments, we have the following requirements on the of the type tct of the type constructor and the types dct_j of the data constructors:

(Below we use the following notation: $\overrightarrow{tca} = [tca_1, \ldots, tca_{\#tca}]$,
$\overrightarrow{dca_j} = [dca_{j,1}, \ldots, dca_{j,\#dca_j}]$, etc. If $\overrightarrow{a} = [a_1, \ldots, a_n]$ and $\overrightarrow{A} = [A_1, \ldots, A_n]$ then we use $\Pi\overrightarrow{a} : \overrightarrow{A}.B$ to denote $\Pi a_1 : A_1. \ldots \Pi a_n : A_n.B$.)

$\exists \overrightarrow{tcat}, \overrightarrow{dcat} \in E$:

- $tct = \Pi\overrightarrow{tca} : \overrightarrow{tcat}.\star$

  for every $1 \leq j \leq \#dca$:
- $dct_j = \Pi\overrightarrow{tca} : \overrightarrow{tcat}.\Pi\overrightarrow{dca}_j : \overrightarrow{dcat}_j.tc\ \overrightarrow{tca}$
- $\vdash dc_j : dct_j : \star$

Note that the use of the dependent product (Π) in our definition to 'link' the data constructor arguments makes it possible to let the types of data constructor arguments depend on other data constructor arguments. In the Haskell definition of ADTs (see definition 3.1) this is not possible, because there the arrow ($\rightarrow$), which can be seen as an independent product, is used to 'link' the data constructor arguments. Because of the same reason it is hard to introduce ADTs in PTS using 'sum' and 'product' types.

In the definition of a PTS with ADTs below we use the same names as above but we add an extra index i to keep the different introduced ADTs apart.

Definition 3.2 (PTSs with ADTs)
A PTS with ADTs is a tuple $(P, ADTS)$ where:

1. P is a Pure Type System, let V, E be the sets of variables and expressions of P.
2. $ADTS = [ADT_1, \ldots, ADT_{\#ADT}]$ with
   (a) for every $1 \leq i \leq \#ADT$:
   $ADT_i = [tc_i : tct_i, dc_{i,1} : dct_{i,1}, \ldots, dc_{i,\#dc_i} : dct_{i,\#dc_i}]$,
   (b) for every $1 \leq i \leq \#ADT$, $1 \leq j \leq \#dc_i$:
   $tc_i, dc_{ij} \in V$ and $tct_i, dct_{ij} \in E$.

   such that for every $1 \leq i \leq \#ADT$:

   - $tct_i = \Pi\overrightarrow{tca}_i : \overrightarrow{tcat}_i.\star$

     for every $1 \leq j \leq \#dc_i$:
   - $dct_{ij} = \Pi\overrightarrow{tca}_i : \overrightarrow{tcat}_i.\Pi\overrightarrow{dca}_{ij} : \overrightarrow{dcat}_{ij}.(tc_i\ \overrightarrow{tca}_i)$
   - $\overrightarrow{tc} : \overrightarrow{tct} \vdash dc_{ij} : dct_{ij} : \star$

The set of expressions, the set of contexts and the reduction relation of PTSs with ADTs are defined in the same way as for PTS without ADTs. Of course, there is a different typability relation.

Definition 3.3 (Typability in a PTS with ADTs)
Let $(P, ADTS)$ be a PTS with ADTs where ADTS is of the form as in definition 3.2:

Define $\Sigma = [c : ct | c : ct \leftarrow ADT, ADT \leftarrow ADTS]$

We say that $a : A$ can be derived from context Γ in $(P, ADTS)$, notation $\Gamma \vdash_{(P,ADTS)} a : A$, if-and-only-if $\Sigma \mathbin{+\!\!+} \Gamma \vdash_P a : A$.

Examples of PTSs with ADTs

Example 3.1 (Bool,List)
Let $P = \lambda\omega$ and

$$Bool = [\text{Bool} : \star, \text{True} : \text{Bool}, \text{False} : \text{Bool}],$$
$$List = [\text{List} : \star \to \star. \text{Nil} : \forall a.\text{List } a, \text{Cons} : \forall a.a \to \text{List } a \to \text{List } a]$$
$$ADTS = [Bool, List]$$

then we have:

$$\vdash_{(P,ATDS)} \text{Cons Bool True (Nil Bool)} : \text{List Bool}$$

Example 3.2 (Mutually Recursive Data types)
Let $P = \lambda\omega$ and

$$ADTS = [[A : \star, AC : B \to A], [B : \star, BC : A \to B, S : B]]$$

then we have:

$$\vdash_{(P,ATDS)} AC\ (BC\ (AC\ S)) : A$$

3.2 Case Expressions

In functional programming languages ADTs often come with a special *case* construct. In this section we will examine how we can introduce the case construct in PTSs.

3.2.1 Case Expressions in Haskell

A typical example of the use of the case construct is the following Haskell function.

```
is_empty :: List a -> Bool
is_empty = \xs . case xs of
           Nil        -> True
           Cons y ys  -> False
```

The `case` construct scrutinises its argument (here `xs`) and matches it against the alternatives (here `Nil` and `Cons`). If `xs` is equal to `Nil` the whole expression reduces to `False`, if `xs` is of the form `Cons y ys` the expression reduces to `True`.

3.2.2 Case Expressions in PTSs

In our explicitly typed PTS framework data constructors should be provided with the types they are instantiated at. To denote the empty list of integers, we use Nil Int instead of just Nil. The same applies for the case statement, not only the arguments of the data constructors should be matched against variables, but also the arguments of the type constructor. For instance, a function that duplicates the first element of a list can be defined by:

$$\begin{array}{lcl} \text{dfe} & : & \forall a : \star.\text{List } a \to \text{Bool} \\ \text{dfe} & = & \lambda a : \star, xs : \text{List } a.\text{case } xs \text{ of} \\ & & \{\text{Nil } a \qquad\quad \Rightarrow \quad \text{Nil } a \\ & & ;\text{Cons } a\ y\ ys \quad \Rightarrow \quad \text{Cons } a\ y\ (\text{Cons } a\ y\ ys)\} \end{array}$$

The definition of a PTS with ADTs and case expressions

To extend the PTS frame work with the case construct, we extend the set of expressions with case expressions.

Definition 3.4 (Expressions)
The set *expressions* E_c of a PTS with ADTs and case expressions is formed by the following grammar:

$$\begin{array}{lll} E_c & ::= & V \mid S \mid E_c\, E_c \mid \lambda V : E_c.E_c \mid \Pi V : E_c.E_c \\ & & \mid \text{case } E_c \text{ of } \{[V] \Rightarrow E_c; \ldots ; [V] \Rightarrow E_c\} \end{array}$$

Also the reduction rules are extended:

Definition 3.5 (Reduction relation)
Let $(P, ADTS)$ be a PTS with ADTs where ADTS is of the form as in definition 3.2. The reduction relation $\rightarrow_c$ is the smallest relation on $E_c \times E_c$ such that for every $1 \leq i \leq \#ADT$, $1 \leq j \leq \#dc_i$

$$\text{case } dc\,\overrightarrow{x} \text{ of } \{dc_{(i,j)}\ \overrightarrow{tca}_i\ \overrightarrow{dca}_{(i,j)} \Rightarrow res_j\} \rightarrow_c (\lambda \overrightarrow{tca}_i : \overrightarrow{tcat}_i\, ,\, \overrightarrow{dca}_{(i,j)} : \overrightarrow{dcat}_{(i,j)}.res_j)\ \overrightarrow{x}$$

and closed under the usual rules.

The reduction relation $\rightarrow_{\beta c}$ is defined by

$$\rightarrow_{\beta c} = \rightarrow_\beta \cup \rightarrow_c$$

The typability relation is extended with a rule for case expressions, the (CONV) rule is extended with reduction of case expressions, so that types which contain case expressions can be properly typed.

The conclusion of the (CASE) rule binds e, the dc_j, the $\overrightarrow{tca}_j$, the $\overrightarrow{dca}_j$ and the res_j to the right expressions. The first premise of the (CASE) rule binds the actual type constructor arguments to $\overrightarrow{acta}$. The second premise derives, using the type of the data constructors dc_j and the actual type constructor arguments, the types of the $\overrightarrow{dca}_j$ and binds them to $\overrightarrow{dcat}_j$. The third premise checks whether the types of the right hand sides, instantiated to the actual type constructor arguments, are equal, and if so the result is bound to t. Finally, the fourth premise checks whether the derived type is well formed.

Definition 3.6 (Typability relation)
Let $(P, ADTS)$ be a PTS with ADTs where ADTS is of the form as in definition 3.2. The typability relation $\vdash_c$ on $C \times E_c \times E_c$ is defined by the rules of definition 2.26, extended with:

$$\text{(CASE)}\quad \frac{\begin{array}{c}\Gamma \vdash_c e : tc\ \overrightarrow{atca} \\ \forall j.\Gamma \vdash_c dc_j\ \overrightarrow{atca} : \Pi \overrightarrow{dca}_j : \overrightarrow{dcat}_j.(tc\ \overrightarrow{atca}) \\ \forall j.\Gamma, \overrightarrow{dca}_j : \overrightarrow{dcat}_j \vdash_c res_j[\overrightarrow{tca}_j := \overrightarrow{acta}_j] : t \\ \Gamma \vdash_c t : s\end{array}}{\Gamma \vdash_c \text{case } e \text{ of } \{dc_j\ \overrightarrow{tca}_j\ \overrightarrow{dca}_j \Rightarrow res_j\} : t}$$

and where the (CONV) rule is replaced by:

$$\text{(CONV)}\quad \frac{\Gamma \vdash_c a : A \quad \Gamma \vdash_c B : S \quad A =_{\beta c} B}{\Gamma \vdash_c a : B}$$

Let $\Sigma = [c : ct | c : ct \leftarrow ADT, ADT \leftarrow ADTS]$

We define $\Gamma \vdash_{c(P,ADTS)} a : A$ by

$$\Gamma \vdash_{c(P,ADTS)} a : A \Leftrightarrow \Sigma +\!\!+ \Gamma \vdash_c a : A$$

The working of the case rule is illustrated by the following instantiation of the rule:

$\Gamma = [\text{List} : \star \to \star . \text{Nil} : \forall a.\text{List } a, \text{Cons} : \forall a.a \to \text{List } a \to \text{List } a, xs : \text{List Int}]$

$$\begin{array}{l}
\Gamma \vdash_c xs : \text{List Int} \\
\Gamma \vdash_c \text{Nil Int} : \text{List Int} \\
\Gamma \vdash_c \text{Cons Int} : \Pi x : \text{Int}.\Pi xs : \text{List Int}.\text{List Int} \\
\Gamma \vdash_c \text{Nil } t[t := \text{Int}] : \text{List Int} \\
\Gamma, x : \text{Int}, xs : \text{List Int} \vdash \text{Cons } t\ x\ (\text{Nil } t)[t := \text{Int}] : \text{List Int} \\
\Gamma \vdash_c \text{List Int} : \star \\
\hline
\begin{array}{lll}
\Gamma \vdash_c & \text{case } xs \text{ of} & \\
 & \{\text{Nil } t & \Rightarrow\ \ \text{Nil } t \\
 & ;\text{Cons } t\ x\ xs & \Rightarrow\ \ \text{Cons } t\ x\ (\text{Nil } t)\} : \text{List Int}
\end{array}
\end{array}$$

Examples of the use of case expression in extended PTSs

Example 3.3 (Existential Data Types)

This example shows that existential data types, in the form as introduced by Laufer in [14], come 'for free' in our extended PTSs. In Laufer's construction we consider data constructor arguments of sort type as existentially quantified. For example in the ADT: $[E : \star, \text{EC} : \forall a : \star.a \to (a \to \text{Bool}) \to \text{E}]$, we say that a is an existentially quantified variable. We consider a to be existentially quantified because when we have an expression of type E, say $e : \text{E}$, then we know that there *exists* a type a such that $e = \text{EC}\ a\ x\ f$ with $x : a$ and $f : a \to \text{Bool}$. Although we do not know which a we are dealing with, we do know that we can apply f to x which results in an expression of type Bool.

The idea behind existential polymorphism is that a term of an existential type is like an object with some data being 'private', and not available for external manipulations, and other data being 'public' and available for external use.

We illustrate the use of 'existentials' below.

$$\begin{array}{lcl}
P & = & \lambda\omega \\
Ex & = & [E : \star, \text{EC} : \forall a : \star.a \to (a \to \text{Bool}) \to \text{E}], \\
List & = & [\text{List} : \star \to \star.\text{Nil} : \forall a.\text{List } a, \text{Cons} : \forall a.a \to \text{List } a \to \text{List } a] \\
Nat & = & [\text{Nat} : \star, \text{Zero} : \text{Nat}, \text{Succ} : \text{Nat} \to \text{Nat}] \\
ADTS & = & [Ex, List, Nat]
\end{array}$$

$$\begin{array}{lcl}
\text{isEmpty} & : & \forall a.\text{List } a \to \text{Bool} \\
\text{isEmpty} & = & \lambda a : \star, \lambda xs : (\text{List } a). \\
 & & \text{case } xs \text{ of} \\
 & & \{\text{Nil } t \qquad\quad \Rightarrow\ \ \text{True} \\
 & & ;\text{Cons } t\ x\ xx\ \ \Rightarrow\ \ \text{False} \\
 & & \}
\end{array}$$

$$\begin{array}{lcl}
\text{apply} & : & \text{E} \to \text{Bool} \\
\text{apply} & = & \lambda e : E. \\
 & & \text{case } e \text{ of} \\
 & & \{\text{EC } t\ x\ f\ \ \Rightarrow\ \ f\ x \\
 & & \}
\end{array}$$

then we have:

$\vdash_{c(P,ATDS)}$ EC (List Int) (Nil Int) (isEmpty Int) : E
$\vdash_{c(P,ATDS)}$ EC Bool False (id Bool) : E
$\vdash_{c(P,ATDS)}$ apply (EC Bool False (id Bool)) : Bool
$\vdash_{c(P,ATDS)}$ apply (EC (List Int) (Nil Int) (isEmpty Int)) : Bool
apply (EC Bool False (id Bool)) $\twoheadrightarrow_{\beta c}$ False
apply (EC (List Int) (Nil Int) (isEmpty Int)) $\twoheadrightarrow_{\beta c}$ True

In Laufer's construction existentially quantified variables are not allowed to escape the scope of the quantifier, as in the following function:

$$\begin{array}{lll} \text{apply}' & = & \lambda e : EC. \\ & & \text{case } e \text{ of} \\ & & \{\text{EC } t\ x\ f \quad \Rightarrow \quad x \\ & & \} \end{array}$$

The reader can check that the fourth premise of the (CASE) rule, which demands that the derived type is typable, is not met. Therefore this expression is indeed not typable in our extended PTS.

Example 3.4 (Dependent Types)

In this example we illustrate the use of dependent types; types that depend on terms. We give an implementation of nzip, a n-ary zip function, that is a function that for arbitrary n zips n lists together. The function takes as first argument a list of types, and as second argument a nested product of lists of those types. For example, if the first argument is [Int, Bool, Nat] (we use a sugared notation for the list of types here), then the type of the second argument is (List Int, (List Bool, List Nat)) (again we use a sugared notation, this time for products, leaving out some of the explicit typing). Because the second (type-)argument depends on the first (term-)argument we need dependent types to describe this function. Suppose the second argument is ([1, 2], ([True, False], [Zero, Zero])), then the output of the function will be [(1, (True, Zero)), (2, (False, Zero))]

First we introduce the list of types ADT:

$$[\text{ListT} : \star, \text{OneT} : \star \to \text{ListT}, \text{ConsT} : \star \to \text{ListT} \to \text{ListT}]$$

and the pair ADT:

$$[\text{PairT} : \forall a : \star, b : \star.\star, \text{Pair} : \forall a : \star, b : \star : a \to b \to \text{PairT}\ a\ b]$$

The mapT function defines a map over a list of types.

$$\begin{array}{lll} \text{mapT} & : & (\star \to \star) \to \text{ListT} \to \text{ListT} \\ \text{mapT} & = & \lambda f : (\star \to \star).\lambda ts : \text{ListT}. \\ & & \text{case } ts \text{ of} \\ & & \{\text{OneT } t \quad \Rightarrow \quad \text{OneT } (f\ t\ ) \\ & & ;\text{ConsT } t\ r \quad \Rightarrow \quad \text{ConsT } (f\ t)\ (\text{mapT } f\ r)\} \end{array}$$

The prodT function maps a list of types into a product of types.

$$\begin{array}{lll} \text{prodT} & : & \text{ListT} \to \star \\ \text{prodT} & = & \lambda ts : \text{ListT}. \\ & & \text{case } ts \text{ of} \\ & & \{\text{OneT } t \quad \Rightarrow \quad t \\ & & ;\text{ConsT } t\ r \quad \Rightarrow \quad \text{PairT } t\ (\text{prodT } r)\} \end{array}$$

To define nzip we need the normal zip function. Because its implementation is not interesting, we only give its type.

$$\text{zip} \; : \; \Pi a : \star, b : \star.\text{List } a \to \text{List } b \to \text{List } (\text{PairT } a\ b)$$

nzip recursively breaks down the list of types ts and the product of lists xs. Note that the outer most case expression has a explicit type signature. The reason for this is the following. The xs on the right hand side of the OneT _ has type prodT (mapT List ts). Assuming that ts if of the form OneT _ it can be derived that prodT (mapT List ts) $=_\beta$ List (prodT ts). So, in that case the xs on the right hand side of OneT _ also has the type List (prodT ts). However the information that ts is of the form OneT _ is not taken into account by the (CASE) rule. Because the scrutinezed expression can be of arbitrary complexity it is not in general possible to take the consequences of the form of the scrutenized expression into account. So, with dependent types, the typing rules cannot always deduce the type of case expressions, therefore we allow, just as in Cayenne [1], case expressions to be explicitly typed.

$$\begin{array}{lcl}
\text{nzip} & : & \Pi ts : \text{ListT}.\text{prodT } (\text{mapT List } ts\ ) \to \text{List } (\text{prodT } ts) \\
\text{nzip} & = & \lambda ts : \text{ListT}. \\
 & & \lambda xs : \text{prodT } (\text{mapT List } ts). \\
 & & \text{case } ts \text{ of} \\
 & & \{\text{OneT } \_ \quad\;\; \Rightarrow \quad xs \\
 & & ;\text{ConsT } t'\ ts' \;\; \Rightarrow \quad \text{case } xs \text{ of} \\
 & & \qquad\qquad\qquad\qquad \{\text{Pair } \_\ \_\ xs'\ pxs' \quad \Rightarrow \quad \text{nzip } t'\ (\text{prodT } ts')\ xs'\ (\text{nzip } ts'\ pxs')\} \\
 & & \} : \text{List } (\text{prodT } ts)
\end{array}$$

3.3 Definitions

In most functional programming languages a *program* consists of number of *definitions*. The purpose of a definition is to introduce a *binding* associating a given *variable* with a given *term*. The variable can be used in the rest of the program as an abbreviation referring to the term.

In this section we will explain how we can extend the PTS framework to deal with definitions.

3.3.1 Definitions in Haskell

We will illustrate the use of definitions by the Haskell program below

```
double :: Int -> Int
double = \n . n + n

quadruple :: Int -> Int
quadruple = \n . double n + double n
```

in this program the variable `double` in the definition of `quadruple` refers to the definition of `double` above.

The effect of the definition of `double` is twofold. First of all, it introduces a new type binding: `double :: Int -> Int`, that will be used in the type checking process. Furthermore, the definition affects reduction. For instance `double 3` will reduce to `(\n . (n + n)) 3`.

3.3.2 Definitions in PTSs

As illustrated in the Haskell example above, the introduction of definitions affects both the typability and the reduction relation. This is reflected in the definitions below.

Definition 3.7 (PTSs with ADTs and definitions.)
A PTS with ADTs and definitions is a triple PAD = (P, ADTS, DS), where $(P, ADTS)$ is a PTS with ADTs and $DS = [D_1, \ldots, D_{\#ds}]$ is a list of definitions with

- for every $1 \leq i \leq \#ds : D_i = (v_i : t_i, e_i)$ with $v_i \in V$, $t_i, e_i \in E$,
- $\Delta = [v_1 : t_1, \ldots, v_{\#ds} : t_{\#ds}]$.

such that for every $1 \leq i \leq \#ds$:

1. there exist an $s \in S$ s.t. $[v_1 : t_1, \ldots, v_{i-1} : t_{i-1}] \vdash_{c\delta(P,ADTS)} t_i : s$
2. $\Delta \vdash_{c\delta(P,ADTS)} e_i : t_i$

The first clause demands that types of defined variables should be well formed. Because type t_i may contain the defined variables $v_1, \ldots, v_{i-1}$ their type bindings are assumed in the context. The second clause demands that the expression which is referred to by a certain variable has indeed the type of that variable. Because both the expressions e_i and their types t_i may contain the defined variables $v_1, \ldots, v_{\#ds}$, their type bindings are assumed in the context. The typability relation for PTSs with ADTs and definitions $\vdash_{c\delta(P,ADTS)}$, is defined below.

Definition 3.8 (Reduction relation)
Let $(P, ADTS, DS)$ be a PTS with ADTs where ADTS is of the form as in definition 3.7. The delta reduction relation, denoted $\rightarrow_\delta$, is the smallest relation such that for every $1 \leq i \leq \#ds$:

$$v_i \rightarrow_\delta e_i$$

and closed under the usual rules.

The reduction relation in a PTS with ADTs and definitions, denoted $\rightarrow_{\beta\delta c}$, is defined by:

$$\rightarrow_{\beta\delta c} = \rightarrow_\beta \cup \rightarrow_\delta \cup \rightarrow_c$$

Definition 3.9 (Typability relation)
Let $(P, ADTS, DS)$ be a PTS with ADTs where ADTS is of the form as in definition 3.7. The typability relation $\vdash_{c\delta}$ on $C \times E_c \times E_c$ is defined by the rules of definition 3.6 where the (CONV) rule is replaced by:

$$\text{(CONV)} \qquad \frac{\Gamma \vdash_{c\delta} a : A \quad \Gamma \vdash_{c\delta} B : S \quad A =_{\beta c\delta} B}{\Gamma \vdash_{c\delta} a : B}$$

Let $\Sigma = [c : ct | c : ct \leftarrow ADT, ADT \leftarrow ADTS]$ and $\Delta = [v_1 : t_1, \ldots, v_{\#ds} : t_{\#ds}]$. We define $\Gamma \vdash_{c\delta(P,ADTS)} a : A$ by

$$\Gamma \vdash_{c\delta(P,ADTS,DS)} a : A \Leftrightarrow \Sigma +\!\!+ \Delta +\!\!+ \Gamma \vdash_c a : A$$

Examples of the use of definitions in extended PTSs

Example 3.5 (The map function on lists)
Let:

$$\begin{array}{lcl} P & = & \lambda\omega \\ List & = & [\text{List} : \star \to \star . \text{Nil} : \forall a.\text{List } a, \text{Cons} : \forall a.a \to \text{List } a \to \text{List } a] \end{array}$$

We can define the map function on list as follows. Note that due to explicit typing the map function takes two type arguments.

$$\begin{array}{lllll} (& \multicolumn{4}{l}{map : \forall a : \star, b : \star.(a \to b) \to \text{List } a \to List\ b,} \\ & \multicolumn{4}{l}{\lambda a : \star, b : \star.} \\ & \multicolumn{4}{l}{\lambda f : (a \to b), xs : (\text{List } a).} \\ & \multicolumn{4}{l}{\text{case } xs \text{ of}} \\ & \{\text{Nil } t & \Rightarrow & \text{Nil } b & \\ & ;\text{Cons } t\ x\ xx & \Rightarrow & \text{Cons } b\ \ (fx\ )\ (\text{map } a\ b\ f\ xx) & \\ & \} & & & \\ ) & & & & \end{array}$$

3.4 Undecidability

Unfortunately, the extension of PTSs with ADTs, case expressions and definitions causes undecidability of type checking. The problem lies in the side-condition $A =_{\beta\delta c} A'$ of the (CONV) rule. Checking whether two arbitrary terms are $\beta\delta c$ equivalent is undecidable. The problem can be repaired by making sure that types are strongly normalising, in that case we can just reduce A and A' to normal form, and check whether the normal forms are equal. So, we can replace the side-condition by $\exists N \in E : A \twoheadrightarrow_{\beta\delta c} N$, $A' \twoheadrightarrow_{\beta\delta c} N$. To make sure that types are strongly normalising we have to forbid dependent types and (general) recursion on the level of types. In section 4 we prove that a system constructed in this way has a decidable type checking problem.

3.5 Operational Semantics

The *operational semantics* of a functional programming language defines the way expressions are reduced. Part of the operational semantics is defined by the reduction rules of our language. The reduction rules do not specify which subexpression should be reduced when multiple subexpressions of a term are reducible. Consider, for instance, in a PTS with the definition (id : Int $\to$ Int, $(\lambda n : \text{Int}.n)$), the expression:

$$(\lambda x : \text{Int}.\lambda y : \text{Int}.y)(\text{id } 3)(\text{id } 5)$$

We can reduce this expression to normal form by in different ways. For instance via

$$\begin{array}{ll} (\lambda x : \text{Int}.\lambda y : \text{Int}.y)\ (\text{id } 3)\ (\text{id } 5) & \to_\delta \\ (\lambda x : \text{Int}.\lambda y : \text{Int}.y)\ ((\lambda n : \text{Int}.n)\ 3)\ (\text{id } 5) & \to_\beta \\ (\lambda x : \text{Int}.\lambda y : \text{Int}.y)\ 3\ (\text{id } 5) & \to_\delta \\ (\lambda x : \text{Int}.\lambda y : \text{Int}.y)\ 3\ ((\lambda n : \text{Int}.n)\ 5) & \to_\beta \\ (\lambda x : \text{Int}.\lambda y : \text{Int}.y)\ 3\ 5 & \to_\beta \\ (\lambda y : \text{Int}.y)\ 5 & \to_\beta \\ 5 & \end{array}$$

but also via:

$$\begin{array}{ll}
(\lambda x : \mathrm{Int}.\lambda y : \mathrm{Int}.y)\ (\mathrm{id}\ 3)\ (\mathrm{id}\ 5) & \rightarrow_\beta \\
(\lambda y : \mathrm{Int}.y)\ (\mathrm{id}\ 5) & \rightarrow_\beta \\
\mathrm{id}\ 5 & \rightarrow_\delta \\
(\lambda n : \mathrm{Int}.n)\ 5 & \rightarrow_\beta \\
5 &
\end{array}$$

A reduction strategy defines which subexpression of a term should be reduced when multiple subexpressions are reducible. A reducible subexpression is called a *redex*.

Definition 3.10 (Reduction Strategy)

A reduction strategy F is a map $F : E_c \rightarrow E_c$ such that $M \rightarrow_{\beta\delta c} F(M)$ for all $M \in E$.

Definition 3.11 (Redeces)

Let $(P, ADTS, DS)$ be a PTS with ADTs, case expressions and definitions, of the form as in definition 3.7.

A (sub)term M is called a

- a β-redex, if M is of the form $(\lambda x : A.N)\ M$,
- a case redex, if M is of the form case $dc\ \overrightarrow{N}$ of $\{\ldots\}$, with $dc \in ADTS$,
- a δ-redex, if M is of the form v_i, with $(v_i, e_i) \in DS$.

We have chosen the normal order reduction strategy for evaluating expressions. This strategy says that iteratively the *left-most outer-most* redex should be reduced. We have chosen for this reduction strategy because it is a *lazy* strategy which means that redeces are only reduced if their result is needed to achieve a normal form. (We do not associate the term lazy with sharing of objects in memory.) Lazy strategies are attractive because they make it possible to work with infinite data structures. It is outside the scope of thesis to go into all the differences between lazy and non-lazy languages, we refer the interested reader to [19].

Definition 3.12 (Left-Most Outer-Most Redex)

Let $N \in E$ and let $\overrightarrow{R} \in E$ be redeces of N. We say that $R \in \overrightarrow{R}$ is an outer-most redex, if there exist no $R' \in \overrightarrow{R}$ such that R is a subexpression of R'. We say that R is the left-most outer-most redex of E if there is no outer-most redex $R'' \in \overrightarrow{R}$ such that R'' is to the left of R in the tree representation of N.

Definition 3.13 (Normal Order Reduction Strategy)

The normal order reduction strategy is defined by $F(M) = M'$ where M' is the result of reducing the left-most outer-most redex of M. If M is normal-form than $M' = M$.

An example of a reduction that uses the left-most outer-most strategy is the second reduction of $(\lambda x : \mathrm{Int}.\lambda y : \mathrm{Int}.y)\ (\mathrm{id}\ 3)\ (\mathrm{id}\ 5)$ above.

Definition 3.14 (Operational Semantics)

The operational semantics of our extended language is defined by the β-,δ- and case-reduction rules and the normal order reduction strategy.

Chapter 4

Type Checking PTSs

In this chapter we investigate a type checker for the language introduced in the previous chapter. First, we study Barthe's type checking algorithm for the class of so-called *injective* PTSs. Then, we investigate how his algorithm can be modified to deal with our extended language.

4.1 Barthe's algorithm

4.1.1 Introduction

Many interesting PTSs have decidable type checking. For those systems the question arises whether there exist efficient type checking algorithms. Often type checking algorithms are based on a set of *syntax-directed* rules. Informally, a set of rules is syntax directed if there is at most one way to derive a type for a given term M in a given context Γ and the type is unique. The typing rules for PTSs are not syntax directed, in particular because the (CONV) rule can be applied at any moment in a derivation.

One way of constructing a type directed set of rules for PTSs is to 'distribute' the (CONV) rule over the other rules. In this way we arrive Robert Pollack's syntax directed rules for PTSs[25]. Unfortunately, the completeness of these rules is an open problem up till now. The main problem in proving the completeness of Pollack's rules lies in the second premise of the (LAM) rule $(\Pi x : A.B) : s$. When trying to prove the completeness of the rules by induction on the derivations, the induction step for the (LAM) rule cannot be completed, precisely because of its second premise. A full analysis of the problems with a proof of the completeness of Pollack's rules lies beyond the scope of this paper, see [20].

Barthe's [4] solution to Pollack's problem is to formulate a new (LAM) rule, based on the so-called classification algorithm, and to distribute the (CONV) rule over this new set of rules. Barthe's rules give a sound and complete syntax directed system for the class of injective PTSs. Below we will introduce the concepts of injectivity, the classification algorithm and Barthe's classification based rules.

4.1.2 Injective PTSs

A PTS is functional if given a sort s_1 there is at most one sort s_2 such that (s_1, s_2) is an axiom, and if given sorts s_1, s_2 there is at most one sort s_3 such that (s_1, s_2, s_3)

is a rule. The definition of injective PTSs is a slight variation this theme.

Definition 4.1 (Injective)
Let $P = (S, A, R)$ be a PTS.

- We say P is functional if for every $s_1, s_2, s_2', s_3, s_3' \in S$:
$$\begin{array}{llll} (s_1, s_2) \in A & \& \quad (s_1, s_2') \in A & \Rightarrow & s_2 = s_2' \\ (s_1, s_2, s_3) \in R & \& \quad (s_1, s_2, s_3') \in R & \Rightarrow & s_3 = s_3' \end{array}$$

- We say P is injective if it is functional and for every $s_1, s_2, s_2', s_3, s_3' \in S$:
$$\begin{array}{llll} (s_1, s_2) \in A & \& \quad (s_1', s_2) \in A & \Rightarrow & s_1 = s_1' \\ (s_1, s_2, s_3) \in R & \& \quad (s_1, s_2', s_3) \in R & \Rightarrow & s_2 = s_2' \end{array}$$

Theorem 4.1 (Injectivity of the λ-cube)
All the systems of the λ-cube are injective.

Proof: Easy.

In an injective PTSs, given a sort s_1, there is at most one sort s_2 such that (s_1, s_2) is an axiom, and therefore we can define a function from sorts to sorts which, given a sort s_1 yields either the unique sort s_2 such that (s_1, s_2) is an axiom or $\uparrow$ (denoting undefined), if no such s_2 exists. For injective PTSs we can define a number of such mappings.

Definition 4.2 ($.^-$, $.^+$, ρ, μ)
For every set A, we let $A^\uparrow$ denote the set $A \cup \{\uparrow\}$. If $f \in A \rightarrow B^\uparrow$ and $a \in A$, we write $f\ a \downarrow$ to denote $f\ a \neq \uparrow$.

- The map $.^- : S^\uparrow \rightarrow S^\uparrow$ is defined by:
$$s^- = \begin{cases} s' & \text{if } (s', s) \in A \\ \uparrow & \text{otherwise} \end{cases}$$

- The map $.^+ : S^\uparrow \rightarrow S^\uparrow$ is defined by:
$$s^- = \begin{cases} s' & \text{if } (s, s') \in A \\ \uparrow & \text{otherwise} \end{cases}$$

- The map $\rho : S^\uparrow \times S^\uparrow \rightarrow S^\uparrow$ is defined by:
$$\rho(s_1, s_2) = \begin{cases} s_3 & \text{if } (s_1, s_2, s_3) \in R \\ \uparrow & \text{otherwise} \end{cases}$$

- The map $\mu : S^\uparrow \times S^\uparrow \rightarrow S^\uparrow$ is defined by:
$$\mu(s_1, s_2) = \begin{cases} s_3 & \text{if } (s_1, s_3, s_2) \in R \\ \uparrow & \text{otherwise} \end{cases}$$

4.1.3 Classification

Injective PTSs form a class of PTSs for which one can define two ‘simple’ functions $\text{sort}(.|.), \text{elmt}(.|.) : C \times E^{\uparrow} \to S^{\uparrow}$ such that:

$$\begin{array}{llll} \Gamma \vdash M : A & \& \quad \Gamma \vdash A : s & \Rightarrow & \text{elmt}(\Gamma|M) = s \\ \Gamma \vdash M : s & \& \quad s \in S & \Rightarrow & \text{sort}(\Gamma|M) = s \end{array}$$

The classification lemma below tells us that for every term in an injective PTS we can compute the type of type of the term by using elmt(.|.). Note that this type of the type of the term is always a sort. For terms whose type is a sort we can compute this sort by using sort(.|.).

Keeping the typing rules of the PTSs framework (definition 2.26), and the definitions of $.^{-}$, $.^{+}$, ρ, μ in mind, all rules, expect for the cases of $\lambda x : A.M$ and $M\ N$ in the definition of the elmt function, are quite easy to understand.

The key to understanding the working of the two more complicated rules is the fact that the last two steps in a derivation of the type of a lambda expression are always (LAM) and (PI), and the last two steps in a derivation of the type of an application are always (APP) and (PI).

Definition 4.3 (Classification Algorithm)
The functions $\text{sort}(.|.), \text{elmt}(.|.) : C \times E^{\uparrow} \to S^{\uparrow}$ are defined as follows:

$$\begin{array}{rcl} \text{sort}(\Gamma|\uparrow) & = & \uparrow \\ \text{sort}(\Gamma|x) & = & (\text{elmt}(\Gamma|x))^{-} \\ \text{sort}(\Gamma|s) & = & s^{+} \\ \text{sort}(\Gamma|M\ N) & = & (\text{elmt}(\Gamma|M\ N))^{-} \\ \text{sort}(\Gamma|\lambda x : A.M) & = & (\text{elmt}(\Gamma|\lambda x : A.M))^{-} \\ \text{sort}(\Gamma|\Pi x : A.B) & = & \rho(\text{sort}(\Gamma|A), \text{sort}(\Gamma, x : A|B)) \\ \\ \text{elmt}(\Gamma|\uparrow) & = & \uparrow \\ \text{elmt}(\Gamma|x) & = & \text{sort}(\Gamma_0|A),\ \text{if } \Gamma = \Gamma_0, x : A, \Gamma_1 \\ \text{elmt}(\Gamma|s) & = & s^{++} \\ \text{elmt}(\Gamma|M\ N) & = & \mu(\text{elmt}(\Gamma|N), \text{elmt}(\Gamma|M)) \\ \text{elmt}(\Gamma|\lambda x : A.M) & = & \rho(\text{sort}(\Gamma|A), \text{elmt}(\Gamma, x : A|M)) \\ \text{elmt}(\Gamma|\Pi x : A.B) & = & (\text{sort}(\Gamma|\Pi x : A.B))^{+} \end{array}$$

Theorem 4.2 (Classification Lemma)
Let $P = (S, A, R)$ be an injective PTS, then:

$$\begin{array}{llll} \Gamma \vdash M : A & \& \quad \Gamma \vdash A : s & \Rightarrow & \text{elmt}(\Gamma|M) = s \\ \Gamma \vdash M : s & \& \quad s \in S & \Rightarrow & \text{sort}(\Gamma|M) = s \end{array}$$

Proof:
See [4].

4.1.4 Classification Based Rules

In this subsection we present Barthe’s *classification based rules*, which give a sound and complete syntax directed system for the class of injective PTSs. In Barthe’s classification based rules the ‘problematic’ premise $(\Pi x : A.B) : s$ in the (LAM) rule is changed to a simpler to calculate premise containing the functions sort(.|.) and elmt(.|.). Furthermore, in Barthe’s rules the (CONV) rule is distributed over the other rules using the notion of weak-head reduction.

Definition 4.4 (Weak-head reduction)

Weak-head reduction $\rightarrow_{wh}$ is the smallest relation such that for every $x \in V$ and $A, P, Q, \overrightarrow{R} \in E$

$$(\lambda x : A.M)\ N\ \overrightarrow{R} \rightarrow_{wh} M[x := N]\ \overrightarrow{R}$$

(Weak-head reduction differs from β-reduction in the sense that weak-head reduction is applied only at the top level of a term)

The reflexive-transitive closure of $\rightarrow_{wh}$ is denoted by $\twoheadrightarrow_{wh}$.

We write $\Gamma \vdash M :\twoheadrightarrow_{wh} A$ for

$$\exists A' \in E.\Gamma \vdash M : A' \text{ and } A' \twoheadrightarrow_{wh} A$$

In the Barthe's classification based rules the 'problematic' premise $(\Pi x : A.B) : s$ in the (LAM) rule is changed to the simpler to calculate premise $\rho(\text{sort}(\Gamma|A), \text{elmt}(\Gamma, x : A|b)) \downarrow$. It is easy to see that $(\Pi x : A.B) : s$ together with the other premise $\Gamma, x : A \vdash b : B$ implies $\rho(\text{sort}(\Gamma|A), \text{elmt}(\Gamma, x : A|b)) \downarrow$: If $(\Pi x : A.B) : s$ then by the (original) (PI) rule we have $\Gamma \vdash A : s_1$; $\Gamma, x : A \vdash B : s_2$ and (s_1, s_2, s) is a rule. We know $s_1 = \text{sort}(\Gamma|A)$, $s_2 = \text{elmt}(\Gamma, x : A|b)$ and because (s_1, s_2, s) is a rule we have $\rho(\text{sort}(\Gamma|A), \text{elmt}(\Gamma, x : A|b)) \downarrow$.

Definition 4.5 (Classification Based Rules)

The typability relation $\vdash_{cl}$ on $C \times E \times E$ for PTSs is defined by the following typing rules:

(axiom) $$\frac{}{[] \vdash_{cl} s_1 : s_2} \qquad (s_1, s_2) \in A$$

(start) $$\frac{\Gamma \vdash_{cl} A :\twoheadrightarrow_{wh} s}{\Gamma, x : A \vdash_{cl} x : A} \qquad x \notin \text{dom}(\Gamma)$$

(weak) $$\frac{\Gamma \vdash_{cl} A : B \quad \Gamma \vdash_{cl} C :\twoheadrightarrow_{wh} s}{\Gamma, x : C \vdash_{cl} A : B} \qquad x \notin \text{dom}(\Gamma)$$

(PI) $$\frac{\Gamma, x : A \vdash_{cl} B :\twoheadrightarrow_{wh} s_2}{\Gamma \vdash_{cl} (\Pi x : A.B) : s_3} \qquad \text{sort}(\Gamma|A) = s_1 \text{ and } (s_1, s_2, s_3) \in R$$

(LAM) $$\frac{\Gamma, x : A \vdash_{cl} b : B}{\Gamma \vdash_{cl} (\lambda x : A.b) : (\Pi x : A.B)} \qquad \rho(\text{sort}(\Gamma|A), \text{elmt}(\Gamma, x : A|b)) \downarrow$$

(APP) $$\frac{\Gamma \vdash_{cl} f :\twoheadrightarrow_{wh} (\Pi x : A'.B) \quad \Gamma \vdash_{cl} a : A}{\Gamma \vdash_{cl} fa : B[x := a]} \qquad A =_\beta A'$$

Theorem 4.3 (Soundness and completeness of $\vdash_{cl}$ w.r.t. $\vdash$)

Let $P = (S, A, R)$ be an injective PTS, then:

$$\begin{array}{lll} \Gamma \vdash_{cl} M : A & \Rightarrow & \Gamma \vdash M : A \\ \Gamma \vdash M : A & \Rightarrow & \exists A' \in E.\Gamma \vdash_{cl} M : A' \ \& A\ =_\beta A' \end{array}$$

Proof:
See [4].

4.2 Extending Barthe's algorithm to our language

In chapter 3 we extended the theory of PTSs with algebraic data types, the case construct and definitions. In this section we extend Barthe's algorithm to deal with these extensions. We add clauses to the classification algorithm to deal with the new expressions, and we extend the typing rules with the rules introduced in the previous chapter.

4.2.1 Extended Classification Algorithm

The rules for n and Int are simple. The sort-rule for case expressions uses the fact that if the type of a case expression is a sort then the type of the case expression is the type of the right hand side of an alternative. The elmt-rule uses the same fact for the type of the type of a case expression.

Definition 4.6 (Extended Classification Algorithm)
The functions $\text{sort}(.|.), \text{elmt}(.|.) : C \times E^{\uparrow} \rightarrow S^{\uparrow}$ are defined by the rules of 4.3, extended with:

$$\begin{array}{rcl}
\text{sort}(\Gamma|n) & = & \uparrow \\
\text{sort}(\Gamma|\text{Int}) & = & \star \\
\text{sort}(\Gamma|\text{case } e \text{ of } \{dc\ \overrightarrow{tca}\ \overrightarrow{dca} \Rightarrow res\}) & = & \text{sort}(\Gamma, \overrightarrow{tca} : \overrightarrow{tcat}, \overrightarrow{dca} : \overrightarrow{dcat}|res) \\
 & & \\
\text{elmt}(\Gamma|n) & = & \star \\
\text{elmt}(\Gamma|\text{Int}) & = & \square \\
\text{elmt}(\Gamma|\text{case } e \text{ of } \{dc\ \overrightarrow{tca}\ \overrightarrow{dca} \Rightarrow res\}) & = & \text{elmt}(\Gamma, \overrightarrow{tca} : \overrightarrow{tcat}, \overrightarrow{dca} : \overrightarrow{dcat}|res)
\end{array}$$

where in the both rules for *case* we have the side condition:

$$dc : \Pi\overrightarrow{tca} : \overrightarrow{tcat}.\Pi\overrightarrow{dca} : \overrightarrow{dcat}.(tc\ \overrightarrow{tca}) \in \Gamma$$

4.2.2 Extended Classification Based Rules

The extended classification based rules are the original classification based rules extended with the rules of the previous chapter.

Definition 4.7 (Extended Classification Based Rules)
Given a PTS with ADTs and definitions $(P = (S, A, R), ADTS, DS)$.

The extended typability relation $\vdash_{ecl}$ on $C \times E_c \times E_c$ is defined by the following typing rules:

(axiom) $$\frac{}{[] \vdash_{ecl} s_1 : s_2} \qquad (s_1, s_2) \in A$$

(start) $$\frac{\Gamma \vdash_{ecl} A :\twoheadrightarrow^{wh}_{\beta\delta c} s}{\Gamma, x : A \vdash_{ecl} x : A} \qquad x \notin \mathrm{dom}(\Gamma)$$

(weak) $$\frac{\Gamma \vdash_{ecl} A : B \quad \Gamma \vdash_{ecl} C :\twoheadrightarrow^{wh}_{\beta\delta c} s}{\Gamma, x : C \vdash_{ecl} A : B} \qquad x \notin \mathrm{dom}(\Gamma)$$

(PI) $$\frac{\Gamma, x : A \vdash_{ecl} B :\twoheadrightarrow^{wh}_{\beta\delta c} s_2}{\Gamma \vdash_{ecl} (\Pi x : A.B) : s_3} \qquad \mathrm{sort}(\Gamma|A) = s_1 \text{ and } (s_1, s_2, s_3) \in R$$

(LAM) $$\frac{\Gamma, x : A \vdash_{ecl} b : B}{\Gamma \vdash_{ecl} (\lambda x : A.b) : (\Pi x : A.B)} \qquad \rho(\mathrm{sort}(\Gamma|A), \mathrm{elmt}(\Gamma, x : A|b)) \downarrow$$

(APP) $$\frac{\Gamma \vdash_{ecl} f :\twoheadrightarrow^{wh}_{\beta\delta c} (\Pi x : A'.B) \quad \Gamma \vdash_{ecl} a : A}{\Gamma \vdash_{ecl} fa : B[x := a]} \qquad A =_{\beta\delta c} A'$$

(CASE) $$\frac{\begin{array}{c}\Gamma \vdash_{ecl} e : tc\ \overrightarrow{atca} \\ \forall j. \Gamma \vdash_{ecl} dc_j\ \overrightarrow{atca} : \Pi \overrightarrow{dca_j} : \overrightarrow{dcat_j}.(tc\ \overrightarrow{atca}) \\ \forall j. \Gamma, \overrightarrow{dca_j} : \overrightarrow{dcat_j} \vdash_{ecl} res_j[\overrightarrow{tca_j} := \overrightarrow{acta_j}] : t \\ \Gamma \vdash_{ecl} t : s\end{array}}{\Gamma \vdash_{ecl} \text{case } e \text{ of } \{dc_j\ \overrightarrow{tca_j}\ \overrightarrow{dca_j} \Rightarrow res_j\} : t}$$

(INT) $$\frac{}{\Gamma \vdash_{ecl} n : \mathrm{Int}} \qquad n \in \mathbb{N}$$

(INT-TYPE) $$\frac{}{\Gamma \vdash_{ecl} \mathrm{Int} : \star}$$

4.2.3 Undecidability

As noted in chapter 3, the extension of PTSs with ADTs, case expressions and definitions causes undecidability of type checking. The rules above are syntax directed, so given an expression an algorithm can always decide which rule to apply. Unfortunately, because checking whether two arbitrary terms are $\beta\delta c$ equivalent is undecidable, an algorithm cannot always decide whether the (APP) rule (with side condition $A =_{\beta\delta c} A'$) is applicable.

The problem can be repaired by making sure that types are strongly normalising. In that case we can just reduce A and A' to normal form, and check whether the normal forms are equal. So, we can replace the side-condition by $\exists N \in E_c : A \twoheadrightarrow_{\beta\delta c} N$, $A' \twoheadrightarrow_{\beta\delta c} N$. To make sure that types are strongly normalising we have to forbid dependent types and (general) recursion on the level of types.

If we change the side condition this way, all side conditions of the rules are decidable. Furthermore, because the rules above are syntax directed, we can prove by an easy induction on the structure of the expression that the type checking process terminates. Therefore the rules above yield a type checking algorithm.

Chapter 5

Implementation

In this section we describe our implementation of a type checker and an interpreter for our language. Because the typechecker uses the interpreter, and not the other way around, we give the implemenatation of the interpreter before we give the implementation of the typechecker. But first of all we introduce the data structures that are used in both the interpreter and the type checker.

5.1 Preliminaries

5.1.1 Abstract Syntax

```
-- The Program
data Program
  = Program [TDecl] [VDecl]

-- Data Type Declarations
data TDecl
 = TDecl TCons [DCons]

type TCons = TVar  -- Type Constructor
type DCons = TVar  -- Data Constructor

-- Value Declarations
data VDecl
 = VDecl TVar Expr
```

A program is a set of type- and value declarations. A type declaration is a pair of a type constructor and a set of data constructors. A value declaration binds a variable to an expression.

```
-- Expressions
data Expr
 = LamExpr TVar Expr         -- Lambda Abstraction
 | PiExpr TVar Expr          -- Pi
 | AppExpr Expr Expr         -- Application
 | CaseExpr Expr [Alt] Expr  -- Case
 | VarExpr TVar              -- Typed Variable
 | LitExpr Lit               -- Literal
 | SortExpr Sort              -- Sorts
 | Unknown
```

The expression data type closely follows the abstract syntax of PTSs. A new clause `Unknown` is introduced for unknown types.

```
-- Typed Variables
data TVar
 = TVar Var Expr

-- Variables
data Var
 = Var String
 | Anonymous
```

In our implementation every variable is annotated with its type. We will explain the advantages of this approach in the next section.

```
-- Case Alternatives
data Alt
 = Alt TCons [TCA] [DCA] Expr

type TCA = TVar -- type constructor argument
type DCA = TVar -- data constructor argument
```

The case alternative data type follows the definition in the chapter 3.

```
-- Literals
data Lit
 = LitInt  Integer
 | IntType
```

There are two literals: integers and the integer type.

```
-- Sorts
data Sort
 = Star
 | Box
 | SortNum Integer
```

In the sort definition there are special clauses for $\star$ and $\Box$. Other sorts can be introduced by enumeration.

5.1.2 Delta Rules

Definitions in our language (defined in 3.7) are internally represented as a list of *delta rules*. A delta rule is is a tuple of a variable and an expression. The meaning of a delta rule of form `DeltaRule v e` is that the variable `v` reduces to the expression `e`.

```
data DeltaRule       = DeltaRule TVar Expr
type DeltaRules      = [DeltaRule]
```

The link between a program and its list of delta rules is straightforward.

```
prog2DeltaRules :: Program -> DeltaRules
prog2DeltaRules (Program _ vdecls) = map vDecl2DeltaRule vdecls

vDecl2DeltaRule :: VDecl -> DeltaRule
vDecl2DeltaRule (VDecl tv ex)  = DeltaRule tv ex
```

5.1.3 Substitutions

A main ingredient of both the interpreter and the type checker is *substitution*. The function `applySubst` performs substitution, it can be applied to values whose type is in the class `SubstC`.

```
data SSubst         = Sub TVar Expr -- singleton substitution
type Subst          = [SSubst]

class SubstC t where
  applySubst   :: Subst  -> t -> t
```

5.2 Implementing an Interpreter

The operational semantics of our language has been formalised in section 3.5. In this section we give an implementation of the operational semantics in the form of an interpreter. An interpreter is a function which given a program reduces the main term of this program to a certain normal form. The implementation of the interpreter is divided into two parts, the first part implements the reduction of single redeces, the second part implements the reduction of complete expressions using the normal order reduction strategy.

5.2.1 Reducing Redeces

Recall from 3.5 that, given a PTS with ADTs and definitions, (P, AD, DS), a β-redex is an expression of the form $(\lambda x : A.M)\ N$, a δ-redex is an expression of the form v with $(v, e) \in DS$, and a case-redex is an expression of the form case $dc\ \overrightarrow{x}$ of ... with dc a data constructor.

The function `isRedex` checks whether an expression is a redex. If the expression is indeed a redex the function tells us whether the expression is a β, δ or case redex. If the expression is a delta redex the `isRedex` function also returns the delta rule which matches the expression.

Actually, `isRedex` does not completely comply to the definition of case redeces. For every expression of the form `CaseExpr ex _ _`, `isRedex` returns `CaseRedex`, while only case expressions of which the left most sub expression of `ex` is a data constructor are case redeces. We have chosen for this deviation because checking whether something is indeed a genuine case redex would be quite costly in terms of performance.

```
data RedexInf = NoRedex
              | BetaRedex
              | CaseRedex
              | DeltaRedex DeltaRule

isRedex :: DeltaRules -> Expr -> RedexInf
isRedex deltaRules expr
 = case expr of
    AppExpr (LamExpr _ _) _  -> BetaRedex
    CaseExpr _ _ _           -> CaseRedex
    VarExpr tv               -> case lookup'' tv deltaRules of
                                    Just deltarule  -> DeltaRedex deltarule
                                    Nothing         -> NoRedex
    _                        -> NoRedex


lookup'' :: TVar -> DeltaRules -> Maybe DeltaRule
```

reduceRedex reduces a redex expression, given information about this redex. It matches against the information in the ri argument and calls the appropriate reduce function.

```
reduceRedex :: DeltaRules -> RedexInf -> Expr -> Expr
reduceRedex dr ri ex = case ri of
 BetaRedex      -> reduceBeta ex
 CaseRedex      -> reduceCase dr ex
 DeltaRedex dr1 -> reduceDeltaRule ex dr1
```

reduceBeta simply performs a substitution as defined for β-reduction in definition 2.24.

```
reduceBeta :: Expr -> Expr
reduceBeta (AppExpr  (LamExpr tv ex1) ex2)
 = applySubst [(Sub tv ex2)] ex1
```

Given an expression of the form `CaseExpr ex alts _`, `reduceCase` reduces `ex` to its weak head normal form `whnf`, using `reduce_to_whnf` which is defined below. Reducing `ex` to its weak head normal form ensures that the left most sub expression is a data constructor. The `reduceBeta` function matches `whnf` against the alternatives using `lookupA`. A lambda expression with as body the right hand side of the matching alternative will be substituted for the left most sub expression of the weak head normal form, as defined in definition 3.5.

```
reduceCase :: DeltaRules -> Expr -> Expr
reduceCase dr (CaseExpr ex alts _) =
 case lookupA whnf alts of
  Just (Alt tc tcas dcas res) ->
   applySubToLeftMost [Sub tc (foldr LamExpr res (tcas++dcas))] whnf
  Nothing                     ->
   error $ "runtime error: missing alternative in case expression"
 where whnf = reduce_to_whnf dr ex

lookupA :: Expr -> [Alt] -> Maybe Alt
applySubToLeftMost :: Subst -> Expr -> Expr
```

`reduceDelta` implements definition 3.8. Because the matching delta rule is given, the function simply returns the second argument of the rule.

```
reduceDelta :: Expr -> DeltaRule -> Expr
reduceDelta _ (DeltaRule _ ex2) = ex2
```

5.2.2 Reducing Expressions

In the previous subsection we described our implementation of the reduction of redeces. In this subsection we will look at the reduction of expressions. Because an expression may contain more than one redex we need to make a choice in which order the redeces of an expression are reduced. The order in which redeces are reduced is called a reduction strategy. (see section 3.5.)

We have chosen the normal order reduction strategy for evaluating expressions. This strategy says that itteratively the *left-most* redex should be reduced (see section 3.5 for the reasons we chose this strategy.).

When evaluating expressions the final result should be a instance of an algebraic data type or a constant, and not a function. That means that redeces inside a λ-expression do not have to be reduced.

The function `eval` reduces expressions using the normal order reduction strategy, while neglecting redeces inside λ-expressions. `eval` can be implemented more efficient if the result of the `eval` function applied to an expression tells whether or not a reduction has taken place. Because of this reason `eval` applied to a list of delta rules and an expression yields a value of type `Maybe Expr`. If `medium` has reduced any redeces of the expression it will ouput a value of the form `Just exr`, where `exr` is the reduced expression, otherwise it will output `Nothing`.

If the expression at hand is a redex `eval` reduces it, and recursively tries to reduce it again. If the expression is an application, `eval` calls `evalApp`. `evalApp` first tries to reduce the left sub expression, if that succeeds it recursively calls `eval` upon the whole expression, if the left sub expression can not be reduced it will try to reduce the right sub expression. If this it is possible `eval` will return the whole expression, note that in this case no recursive call is needed.

```
eval :: DeltaRules -> Expr -> Maybe Expr
eval dr ex
 | redexinf/=NoRedex                = mplus (eval dr reduced)
                                            (Just reduced)
 | isApp  ex                        = evalApp dr ex
 | otherwise                        = Nothing
 where redexinf = isRedex dr ex
       reduced  = (reduceRedex dr redexinf ex)


evalApp dr ex@(AppExpr ex1 ex2) =
 case (eval dr ex1) of
   Just ex1r -> mplus (eval dr (AppExpr ex1r ex2))
                            (Just (AppExpr ex1r ex2))
   Nothing   -> case (eval dr ex2) of
                 Just ex2r   -> Just $ AppExpr ex1 ex2r
                 Nothing     -> Nothing
```

The function `eval` reduces expressions using the normal order strategy and stops when it reaches a normal form or a lambda expression. Next to `eval` we need two other reduction functions: a function `strong` which reduces expressions to a normal form (and does not stop when it reaches a lambda expression). `strong` is used in the implementation of the (APP) rule, see section 4.2.3. Furthermore, we need a function `weak` which reduces expressions to their weak head normal form (see definition 4.4). `weak` is used in the `ReduceCase` function (see above) and the implementation of the $:\twoheadrightarrow_{wh}$ relation in the type checker (see section 5.3.4).

The difference between `weak` and `eval` is that `weak` does not try to reduce the right subexpression in an application.

```
weak :: DeltaRules -> Expr -> Maybe Expr
weak dr ex
 | redexinf/=NoRedex                = mplus (weak dr reduced)
                                            (Just reduced)
 | isApp  ex                        = weakApp dr ex
 | otherwise                        = Nothing
 where redexinf = isRedex dr ex
       reduced  = (reduceRedex dr redexinf ex)

weakApp dr (AppExpr ex1 ex2) =
 case (weak dr ex1) of
   Just ex1r -> mplus (weak dr (AppExpr ex1r ex2))
                      (Just (AppExpr ex1r ex2))
   Nothing   -> Nothing
```

The difference between `strong` and `eval` is that `strong` does not stop when it reaches a lambda expression.

```
strong :: DeltaRules -> Expr -> Maybe Expr
strong dr ex
 | redexinf/=NoRedex                = mplus (strong dr reduced)
                                            (Just reduced)
 | isApp  ex                        = strongApp  dr ex
 | isLam    ex                      = strongLam  dr ex
 | otherwise                        = Nothing
 where redexinf = isRedex dr ex
       reduced  = (reduceRedex dr redexinf ex)

strongApp dr ex@(AppExpr ex1 ex2) =
 case strong dr ex1 of
   Just ex1r -> mplus (strong dr (AppExpr ex1r ex2))
                      (Just $ AppExpr ex1r ex2)
   Nothing   -> case (strong dr ex2) of
                  Just ex2r -> Just $ AppExpr ex1 ex2r
                  Nothing   -> Nothing

strongLam dr (LamExpr (TVar var exv) ex)
 = if
     (isNothing mexvr && isNothing mexr)
   then
     Nothing
   else
     do{exvr<-mplus mexvr (Just exv)
       ;exr <-mplus mexr  (Just ex)
```

```
        ;return $ LamExpr (TVar var exvr) exr
    where
     mexvr = strong dr exv
     mexr  = strong dr ex
```

5.3 Implementing the Type Checker

Implementing the type checker according to the extended classification based rules of definition 4.7 is straightforward. We use a simple write monad to side effect error messages.

5.3.1 Annotating bound variables

In PTSs the binding variables in λ- and Π-expressions are annotated with their type. The type of a bound variable is derived by means of the (var) and (weak) rules.

In our implementation *every* variable is annotated with its type. For example: the expression `\f:Int -> Int.(\n:Int . f n)` will be internally represented as `\f:Int -> Int.(\n:Int . f:(Int -> Int) n:Int)`. This simplifies the design of the type checker; when we annotate every variable with its type the type checker does not need to 'plumb around' an environment, and the implementation of the uninteresting rules (var) and (weak) can separated from the implementation of the interesting rules.

The process of annotating bound variables in programs is carried out by the function `timain` which takes as argument a program and returns the program in which every bound variable is annotated with its type. We refrain from giving the, straightforward, implementation of `timain`, but will illustrate its working with an example.

A variable can be bound in three different ways:

1. By a λ or Π quantifier.
2. By a data type definition.
3. By the left hand side of a case alternative.

In the program below all three forms of bindings are present.

```
data List : * -> *
= { Nil  : \/a:* . List a
  ; Cons : \/a:* . a -> List a -> List a
  }

let  map: (\/a, b. (a -> b) -> List a -> List b)
        = /\a, b.
          \f: (a->b). \xs:(List a).
              case xs of
              { Nil t         => Nil b
              ; Cons t, x, xx => Cons b (f x) (map t b f xx)
              }
```

1. The variable `xs` at the right of the `case` keyword is bound by `\xs:(List a)`, so `xs` will be annotated with the type `List a`.

2. The type of the data constructor `Cons` can be derived from the data type definition of `List`, so all occurrences of `Cons` will be annotated with the type `\/a:* . a -> List a -> List a`.

3. The types of `t,x` and `xx` in the right hand side of the `Cons` alternative can be derived from the type of `Cons`, this results in the annotations: `t:*`, `x:t` and `xx : List t`.

5.3.2 Specifications

Type checking a program is performed with respect to a certain specification of a PTS. The specification data type is given below.

```
type Specification = (Sorts, Axioms, Rules)
type Sorts         = [Sort]
type Axiom         = (Sort,Sort)
type Axioms        = [Axiom]
type Rule          = (Sort,Sort,Sort)
type Rules         = [Rule]
```

Below the specifications of the systems of the lambda cube are implemented.

```
-- lambda cube sorts
lcs :: Sorts
lcs = [Star,Box]

-- lambda cube axioms
lca :: Axioms
lca = [(Star,Box)]

-- the rules of lambda arrow, lambda two,
-- lambda omega and the calc. of constructions
lar,l2r,lor,ccr :: Rules
lar = [(Star,Star,Star)]
l2r = lar ++ [(Box,Star,Star)]
lor = l2r ++ [(Box,Box,Box)]
ccr = lor ++ [(Star,Box,Box)]

-- the specification of lambda arrow, lambda two,
-- lambda omega and the calculus of constructions
la,l2,lo,cc :: Specification
la = (lcs,lca,str)
l2 = (lcs,lca,l2r)
lo = (lcs,lca,lor)
cc = (lcs,lca,ccr)
```

5.3.3 Classification

An important ingredient of Barthe's algorithm is the classification algorithm. Its implementation is straightforward. The sets $E^{\uparrow}$ and $S^{\uparrow}$ are modelled by the types `Maybe Expr` and and `Maybe Sort`.

```
elmt :: Specification -> Maybe Expr -> Maybe Sort
elmt sp me = case me of
 Nothing -> Nothing
 Just e  -> case e of
  VarExpr  (TVar v e)    -> sortt sp $ Just e
  SortExpr s             -> plus sp $ plus sp $ Just s
  AppExpr m n            -> mu sp (elmt sp $ Just n) (elmt sp $ Just m)
  LamExpr (TVar v e) e2  -> rho sp (sortt sp $ Just e)
                                   (elmt sp $ Just e2)
  PiExpr  _ _            -> plus sp $ sortt sp $ Just e
  LitExpr IntType        -> Just $ Star
  LitExpr _              -> Nothing
  CaseExpr _ alts _      -> let ((Alt _ _ _ res):_) = alts in
                            elmt sp $ Just res

sortt :: Specification -> Maybe Expr -> Maybe Sort
sortt sp me = case me of
 Nothing -> Nothing
 Just e  -> case e of
  VarExpr  tv            -> minus sp $ elmt sp $ Just $ VarExpr tv
  SortExpr s             -> plus sp $ Just s
  AppExpr _ _            -> minus sp $ elmt sp $ Just e
  LamExpr _ _            -> minus sp $ elmt sp $ Just e
  PiExpr  (TVar v e) e2  -> rho sp (sortt sp $ Just e)
                                   (sortt sp $ Just e2)
  LitExpr (LitInt _)     -> Just Star
  LitExpr IntType        -> Just Box
  CaseExpr _ alts _      -> let ((Alt _ _ _ res):_) = alts in
                            sortt sp $ Just res
```

5.3.4 The actual type checker

The actual type checker is implemented using a simple writer monad. Errors can be written using the function `addErrorMsg`

```
-- The Type Check Monad
type Error       = String
type Errors      = [Error]

newtype TC t = TC (Errors,t)

instance Monad TC where
 return x    = TC ([],x)
 f >>= g     = TC (let TC (erf,x)     = f
                       TC (erg,y)     = g x
                       in (erf++erg,y))

addErrorMsg :: String -> TC ()
addErrorMsg s = TC([s],())
```

Type checking a PTS is performed with respect to a specification and a reduction relation. In our language the reduction relation is fixed to $\beta-$, case-, and δ-reduction. Whether or not a term can be reduced using $\beta-$ or case-reduction depends only on the term itself. Whether or not a term can be reduced using $\delta-$reduction depends also on the definitions in the rest of the program. Therefore both the specification of a PTS and the delta rules of the program at hand should be available to a function that performs type checking.

We introduce the type `TypeCheck a b` for functions that perform type checking. A function of type `TypeCheck a b` takes as arguments a specification, a list of delta rules and an object of type `a` that should be type checked, the output of the function is of type `TC b`. If `a` is `Expr` than the function returns the type of the expression argument, so in that case `b` is also `Expr`. Otherwise `b` is equal to `()`. In both cases error messages are side effected using the TC monad.

```
-- TypeCheck Type
type TypeCheck a b = Specification -> DeltaRules -> a -> TC b
```

Type checking a program consists of type checking the data types and definitions.

```
-- Program
program :: TypeCheck Program ()
program dr sp (Program tds vds) = do{tds <- mapM (tDecl dr sp) tds
                                    ;vds <- mapM (vDecl dr sp) vds
                                    ;return () }
```

Type checking data type definitions consists of checking whether the given types of the type- and data constructors are correct. Furthermore, it is checked whether the type- and data constructors are of the form of definition 3.2.

```
-- Data Type Declarations
tDecl    :: TypeCheck TDecl ()
tDecl dr sp td@(TDecl tv tvs)
 = do{mapM (\(TVar _ t) -> expr dr sp t) (tv:tvs)
     ;isOfRightForm dr sp td
     ;return ()}


isOfRightForm :: TypeCheck TDecl ()
```

Type checking a definition consists of checking whether the derived type and the given type of the defined expression are equal.

```
-- Value Declarations
vDecl    :: DeltaRules -> Specification -> VDecl -> TC ()
vDecl dr sp (VDecl tv@(TVar _ tv_type) ex)
 = do {ex_type <- expr dr sp ex
      ;if
         not $ equal tv_type ex_type
       then
         do{addErrorMsg $ bindMsg tv tv_type ex ex_type
           ;return $ ()}
       else
         return ()
      }
```

The `expr` function type checks expressions and returns their type. `expr` pattern matches against the structure of the expression and calls the appropriate check function.

```
expr :: TypeCheck Expr Expr
expr dr sp ex = case ex of
 SortExpr _          -> sortExpr dr sp ex                -- (SORT)
 VarExpr _           -> varExpr  dr sp ex                -- (VAR)
 PiExpr _  _         -> piExpr   dr sp ex                -- (PI)
 LamExpr _ _         -> lamExpr  dr sp ex                -- (LAM)
 AppExpr _ _         -> appExpr  dr sp ex                -- (APP)
 CaseExpr _ _ _      -> caseExpr dr sp ex                -- (CASE)
 LitExpr _           -> litExpr  dr sp ex                -- (LIT)
 Unknown             -> return Unknown
```

The (PI) and (APP) rules of definition 4.7 contain the relation $:\twoheadrightarrow_{wh}$, this relation is modelled by the `exprwh` function. The `reduce_to_whnf` function reduces a term to its weak head normal form . (see section 5.2)

```
exprwh :: TypeCheck Expr Expr
exprwh dr sp ex =
 do{ex <- expr dr sp ex
   ;return $ reduce_to_whnf dr ex}
```

The `sortExpr` function follows closely the structure of the (axiom) rule. If there is no axiom for the sort an error message will be returned.

```
sortExpr :: TypeCheck Expr Expr
sortExpr _ (_,a,_) (SortExpr s1)
 =  case lookup s1 a of
    Just s2 -> do{return $ SortExpr s2}
    Nothing -> do{addErrorMsg $ noAxiomMsg s1
                 ;return Unknown}
```

Because every variable is annotated with its type, the `varExpr` function simply returns the annotated type.

```
varExpr :: DeltaRules -> Specification -> Expr -> TC Expr
varExpr _ _ (VarExpr (TVar _ ex)) = return ex
```

The `piExpr` function closely follows the structure of the (PI) rule. Given an expression of the form `PiExpr (TVar v a) b`, the function will first check whether the type of `a` is a sort, if this is the case the sort is stored in `s1`. Then the type of `b` is derived and reduced to weak head normal form. It is checked whether the reduced type of `b` is a sort, if this is the case it is stored in `s2`. Finally, it is checked whether there is a rule of the form `(s1,s2,s3)` in the rules of the current PTS. If this is the case, the type of the expression, `SortExpr s3`, is returned.

```
piExpr :: DeltaRules -> Specification -> Expr -> TC Expr
piExpr dr sp@(_,_,r) piExpr@(PiExpr (TVar v a) b)
 = do{maybe_s1     <- return $ sortt sp (Just a)
```

```
     ;case maybe_s1 of
       Nothing -> do{addErrorMsg $ noSortAMsg piExpr
                    ;return Unknown}
       Just s1  ->
        do{btype        <- exprwh dr sp b
          ;case btype of
            SortExpr s2 ->
             do{case lookup' (s1,s2) r of
                 Nothing       -> do{addErrorMsg $
                                      ruleMsg piExpr s1 s2
                                   ;return Unknown}
                 Just (_,_,s3) -> return $ SortExpr s3
                }
            _           -> do{addErrorMsg $ noSortBMsg piExpr btype
                             ;return Unknown}
          }
      }


lookup' :: (Eq a , Eq b) => (a,b) -> [(a,b,c)] -> Maybe (a,b,c)
```

The lamExpr function closely follows the structure of the (LAM) rule. Given an expression of the form `LamExpr (TVar x a) m`, the function will derive the type of `m` and check whether the side condition of the (LAM) rule holds.

```
lamExpr :: TypeCheck Expr Expr
lamExpr dr sp ex@(LamExpr tv@(TVar x a) m)
 = do{b <- expr dr sp m
     ;case rho sp (sortt sp $ Just a) (elmt sp $ Just m) of
       Just _  -> return $ PiExpr tv b
       Nothing -> do{addErrorMsg $ lamMsg ex
                    ;return Unknown}
     }
```

The function appExpr closely follows the structure of the (APP) rule. First, the types of `f` and `c` are deduced and stored in `f_type` and `a`. Then, it is checked whether `f_type` is of the form `PiExpr (TVar x a') b`. If this is the case `a` and `a'` are reduced to normal form. Finally, if the normal forms of `a` and `a'` are equal the type `b` with `x` substituted for `c` is returned.

```
-- (APP)
appExpr :: TypeCheck Expr Expr
appExpr dr sp (AppExpr f c)
 = do{f_type <- exprwh dr sp f
     ;a      <- expr dr sp c
     ;if
        (not (isPi f_type))
      then
        do{addErrorMsg $ appMsg2 f f_type c a
          ;return  Unknown}
      else
        let
          (PiExpr tv@(TVar x a') b)=f_type
        in
          do{a  <- return $ reduce_to_nf dr a
```

```
            ;a' <- return $ reduce_to_nf dr a'
            ;if
               not (equal a a')
             then
               do{addErrorMsg $ appMsg1 f f_type c a a'
                 ;return Unknown}
             else
               do{return $ applySSubst (Sub tv c) b}
             }
      }
```

The `caseExpr` function closely follows the structure of the (CASE) rule. Given a case expression of the form `CaseExpr ex alts Unknown`, (the `Unknown` argument tells us that the case expression is not explicitly typed as in example 3.4) first the type of the scrutinized expression `ex` is stored in `ex_type`. From this type the type constructor and its actual arguments are derived and stored in `tc` and `atcas`. (For instance if `xs` equals `Cons Int 1 (Nil Int)` then `tc` will equal `List` and `atcas` will equal `Int`.) Then, for every alternative the type of the result expression, with the type constructor arguments substituted with the actual type constructor arguments, is derived. If all these types are equal the common type is returned, if not the `Unknown` type is returned and an error message is side effected.

If the case expression is explicitly typed (as in example 3.4) we just return the type.

```
-- (CASE)
caseExpr :: TypeCheck Expr Expr
caseExpr dr sp ce@(CaseExpr ex alts Unknown) =
 do{ex_type                      <- expr dr sp ex
   ;tc                           <- return $ leftMost ex_type
   ;atcas                        <- return $ ex_atcas ex_type
   ;rest <- mapM (\(Alt _ tcas _ res)->
              do{subst  <- return $ zipWith Sub tcas atcas
                ;res    <- return $ applySubst subst res
                ;expr dr sp res})  alts
   ;ct <- return $ foldr1 (\t1 t2 -> if t1==t2 then t1 else Unknown) rt
   ;if
      ct==Unknown
    then
      do{addErrorMsg $ caseMsg ce
        ;return ct}
    else
      return ct
   }

caseExpr _ _ (CaseExpr _ _ t) = return t
```

The implementation of the rules for the integers is trivial.

```
-- (LIT)
litExpr :: TypeCheck Expr Expr
litExpr _ _ (LitExpr lit)
 = case lit of
   LitInt _ -> do{return $ LitExpr IntType}
   IntType  -> do{return $ SortExpr Star}
```

Chapter 6

Conclusion

We have presented a functional programming language based on PTSs. We have shown how we can define the language by extending the PTS framework with algebraic data types, case expressions and definitions.

Unlike the description of the Henk language in [12] we have given a complete formal definition of the type system and the operational semantics of our language. Another difference between Henk and our language is that our language is defined for a large class of Pure Type Systems, and not only for the systems of the λ-cube.

To be able to experiment with our language we have developed a type checker and an interpreter. The basic ideas behind the interpreter and the type checker are described in the thesis.

We have illustrated that the type system of our language is more powerful than the Hindley-Milner system by showing that a number of meaningful programs that cannot be typed in Haskell can be typed in our language. A real world example of such a program is the mapping of a specialisation of a Generic Haskell function to a Henk program.

We have shown that PTSs are at the top of an hierarchy of increasingly stronger type systems. In functional programming languages based on the systems of this hierarchy the concepts of 'existential types', 'rank-n polymorphism' and 'dependent types' arise naturally. There is no need for ad-hoc extensions to incorporate these features.

Unfortunately PTSs need (at least) explicit typing to have a decidable type checking problem. Because of this reason it is difficult to use our language as a *source* programming language, it would place a heavy burden on a programmer to explicitly type every λ or Π abstraction, and to explicitly give the types at which polymorphic functions should be instantiated.

Therefore, a topic for further research would be how to mix the Hindley-Milner and PTS typing system in one language. This would allow the user to write implicitly typed code when (s)he does not need the strength of PTSs, and to write explicitly typed code for the more advanced components of his/her program.

Another topic of interest is the question exactly how much of the source code should be explicitly typed to keep type checking decidable. A lower bound for such an effort is set in [18] where it is shown that type checking is undecidable for a variant of $\lambda\omega$ in which types can be omitted but a 'marker' must be left where a type has been omitted.

Bibliography

[1] L. Augustsson. Cayenne – a language with dependent types. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* vol. II of Studies in Logic and the Foundations of Mathematics. North-Holland, second edition, 1984.

[3] H. Barendregt. Lambda calculi with types. In G. Abramsky and Maibaum, editors, *Handbook of Logic in Computer Science, vol. II.* Oxford University Press., 1992.

[4] Gilles Barthe. Type-checking injective pure type systems. *Journal of Functional Programming*, 9(6):675–698, November 1999.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] H. Curry. Functionality in combinatory logic. In *Proc. Nat. Acad. Sci, U.S.A.*, volume 20, pages 584–590, 1934.

[7] N. de Bruijn. The mathematical language automath, its usage and some of its extensions. *Lectures Notes in Mathematics*, 125, 1968. Symposium on Automatic Demonstration.

[8] G. de Lavalette. Strictness analysis via abstract interpretation for recursively defined types. *Information and Computation*, 99:154–177, 1992.

[9] J. Girard. *Interpr'etation functionelle et 'elimunation des coupures de l'arithm'etique d'ordre sup'erieur.* PhD thesis, Universit'e Paris VII, 1972.

[10] R. Hinze. Polytypic values possess polykinded types. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, July 2000.

[11] S. Jones and J. editors. Haskell 98 report. http://haskell.org, 1998.

[12] S. Jones and E. Meijer. Henk: a typed intermediate language. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[13] Simon Peyton Jones. Explicit quantification in haskell, http://www.research.microsoft.com/users/simonpj/haskell/quantification.html.

[14] K. Laufer and M. Odersky. An extension of ml with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, 1992.

[15] G. Longo and E. Moggi. Constructive natural deduction and its modest interpretation. *Mathematical Structures in Computer Science*, 1(215254), 1992.

[16] Alastair Reid Mark P Jones. The hugs 98 user manual, http://www.haskell.org/hugs.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[18] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993.

[19] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Graph Rewriting*. Addison-Wesley, 1993.

[20] E. Poll. A typechecker for bijective pure type systems. Technical report, Technical University of Eindhoven, June 1993. CSN93/22.

[21] J. C. Reynolds. Towards a theory of type structure. *Lecture Notes in Computer Science*, 19:157–168, 1974. In Paris Programming Symposium.

[22] Morten Heine. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[23] G.Huet T. Coquand. Huet: Constructions: a higher order proof system for mechanizing mathematics. Technical report, INRIA, May 1985. no. 401.

[24] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Harlow, England, 1996.

[25] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, July 1993.

[26] J. Wells. Typability and type checking in the second-order - calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. Logic Comput. Sci.*, July 1994.