# Pure Type Systems for Functional Programming

[Extended Abstract] *

Jan-Willem Roorda & Johan Jeuring
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089 3508 TB Utrecht, The Netherlands
{jw,johanj}@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. Furthermore, we present an efficient type checking algorithm and an interpreter for this 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.

## 1. INTRODUCTION

One of the important advantages of functional programming languages is that they posses powerful type systems that prevent programmers from writing erroneous programs. If a program contains a type error 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' [14].

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 [12] 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 [14]. The Hindley-Milner system is an implicitly typed system. That means that we do not need -as with an explicitly typed language- to annotate variables in $\lambda$-abstractions with their type. An attractive 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 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, gives a type error. The Haskell type system tries to find 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 -> Int -> Int)` it will be rejected.

Admittedly, a slightly modified version of the above program can be typed by a recent extension [11][9] of Hugs which allows rank-2 types. But also this extension does not provide everything we need: there are still meaningful programs 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 [8, 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 a quantifier to occur.

*A full version of this paper is available at `http://www.cs.uu.nl/~johanj/MSc/jwroorda`

The above examples illustrate that the type system of Haskell (without, but also with extensions) is not strong enough for our current needs. In this article 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 $\lambda$-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 [10] they present a language called 'Henk'. Unfortunately, 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 Peyton Jones, but we have chosen a more formal approach: we will give a 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, whereas Meijer and Peyton Jones only give typing rules for the $\lambda$-cube variants of 'Henk'.

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.

Futhermore, 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 an efficient type checking algorithm for this class of PTSs can be extended to our language.

### *PTSs for Functional Programming: What?*

The contributions of this research are:

- A program language extension for PTSs.
- An efficient type checking algorithm for our language.
- A interpreter for our language.

The implementation of the type checker and the interpreter are available from our website[1]. We enthusiastically welcome experimenting with our language!

[1] `http://www.cs.uu.nl/~johanj/MSc/jwroorda`

$$
\begin{array}{rcll}
E & = & V & \text{(variable)} \\
  & | & E\ E & \text{(application)} \\
  & | & \lambda V : E.E & \text{(abstraction)} \\
  & | & \Pi V : E.E & \text{(product)} \\
  & | & \star & \text{(star)} \\
  & | & \Box & \text{(square)} \\
  \\
V & = & x, y, z \ldots \alpha, \beta, \gamma, \ldots &
\end{array}
$$

**Figure 1: Expressions of the $\lambda$-cube**

### *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, $\lambda$-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 [20]. Classical references for the $\lambda$-calculus and type-theory are [1] and [2].

### *Organisation of this article*

In the next section we introduce the theory of Pure Type Systems. In section three we investigate how the theory of PTSs can be extended with algebraic data types, case expressions and definitions. Then, in section four we study a type checking algorithm for PTSs and investigate how we can extend the algorithm to deal with our extended language. In section five we give the operational semantics of our language. In the last section we conclude and give suggestions for further research.

## 2. PURE TYPE SYSTEMS

The theory of Pure Type Systems is a generalization of the theory of the $\lambda$-cube. Because the $\lambda$-cube is easier to understand than the theory of PTSs we start with introducing the $\lambda$-cube.

### *The $\lambda$-cube*

The $\lambda$-cube is a generalization of a set of eight type-systems including the well known systems $\lambda\!\to$ [5][6], $\lambda 2$ [7][18], $\lambda\omega$ [7], and $\lambda$C [19]. We give a short description of these four systems below.

The system $\lambda\!\to$ is the basis of all type systems for functional programming. In this system it is possible to define terms that depend on other terms, for instance $M \equiv \lambda n : \tau.n$ with $M : \tau \to \tau$. We call $M$ a term depending on term because given a term $N$ of type $\tau$, $M$ applied to $N$, yields another term.

The system $\lambda 2$ extends the system $\lambda\!\to$; every expression that is typable in $\lambda\!\to$ is typable in $\lambda 2$ too. Furthermore, in $\lambda 2$ it is possible to define terms that depend on types, for instance $M \equiv \Lambda\alpha.\lambda x : \alpha.x$ with $M : \forall\alpha.\alpha \to \alpha$. We call $M$ a term depending on a type because $M$ applied to a type yields a term.

The system $\lambda\omega$ extends the system $\lambda 2$. In the system $\lambda\omega$ it is possible to define types that depend on other types, for instance $M \equiv \lambda\alpha : \star.\alpha \to \alpha$ with $M : \star \to \star$. $M$ applied to a type yields a type.

Finally, the system $\lambda$C extends the system $\lambda\omega$. In the system $\lambda$C it is possible to define types that depend on terms (also known as 'dependent types'), for instance $M \equiv \lambda x : \alpha.\alpha$ with $M : \alpha \to \star$. Given a term $N$ of type $\alpha$, $M$ applied to to $N$ yields the type $\alpha$.

The four systems described above each present a new kind of dependency. And although each of these systems has been introduced with its own typing-rules and notations, it it is possible to present the systems in a uniform notation and with a single notion of abstraction parameterized over the permitted dependencies. This is exactly what is done in the $\lambda$-cube!

The set of expressions of the $\lambda$-cube is defined by the grammar given in Figure 1. The first three productions should be familiar to the reader, however it should be noted that the lambda abstraction is explicitly typed. The last two productions, $\star$ and $\Box$, are called sorts. Together with the typing rules, given in Figure 3, they determine which expressions are called types, terms or kinds. We have the following correspondence:

| | | |
|---|---|---|
| $K$ is a kind (in context $\Gamma$) | $\Leftrightarrow$ | $\Gamma \vdash K : \Box$ |
| $T$ is a type | $\Leftrightarrow$ | $\Gamma \vdash T : K : \Box$ |
| $E$ is a term | $\Leftrightarrow$ | $\Gamma \vdash E : T : K : \Box$ |

**Figure 2: The notions of types, terms and kinds.**

So, for instance, we say that $\star$ is a kind because we have $\vdash \star : \Box$.

The fourth production in the expression grammar of the $\lambda$-cube for the (dependent) product ($\Pi$) accounts for the absence of the function arrow ($\to$) and the universal quantifier ($\forall$). The dependent product $\Pi x : A.B$ can be understood as 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 $x$. So, the dependent product subsumes both the arrow and the universal quantifier. For instance, if $f = \lambda n : \text{Int}.n + n$, then $f$ is nothing else than a function from values of type Int to values of type Int, in which the result type Int does not depend on the argument $n$. So, we have $f : (\Pi n : \text{Int}.\text{Int})$ or $f : (\Pi\_ : \text{Int}.\text{Int})$, where _ denotes an anonymous variable. We use $A \to B$ as syntactic sugar for $\Pi\_ : A.B$.

If $g = \lambda\alpha : \star.\lambda x : \alpha.x$, then $g$ is a function from values of type $\star$ to values of type $\alpha \to \alpha$. In this case the result type $(\alpha \to \alpha)$ depends on the value of the argument $\alpha$. So we have $g : (\Pi\alpha : \star.\alpha \to \alpha)$. We use $\forall\alpha : \kappa.B$ as syntactic sugar for $\Pi\alpha : \kappa.B$.

The typing rules of the $\lambda$-cube (given in Figure 3) are parameterized by a set of rules $R$ such that $\{(\star,\star)\} \subseteq R \subseteq \{\star,\Box\} \times \{\star,\Box\}$. We explain below how the elements of the set of rules $R$ correspond with the earlier introduced dependencies between term and types.

The (axiom) rule states that $\star$ is a kind.

The (start) rule states that if $A$ can be typed, we can derive $x : A$ from any context ending with $x : A$.

The (weak) rule allows us to throw away irrelevant bindings.

The (abs) rule allows us to type abstractions. The abstraction $\lambda x : A.b$ is given type $\Pi x : A.B$ if

- given $x : A$ we can derive $b : B$
- $\Pi x : A.B$ can be typed.

The first requirement is natural given the interpretation of the dependent product ($\Pi$). The second requirement restricts the kind of abstractions that can be typed by demanding that the dependent product $\Pi x : A.B$ should be typable. The (pi) rule determines which dependent products are typable.

The (pi) rule allows us to type dependent products. The dependent product $\Pi x : A.B$ is given the type of $B$ if the types of $A$ and $B$ occur in a rule of $R$. For instance $(\Pi\_ : \text{Int}.\text{Int}) = \text{Int} \to \text{Int}$ has type $\star$ because because $\text{Int} : \star$ and $(\star,\star) \in R$.

By taking $(s,t) \in \{\star,\Box\} \times \{\star,\Box\}$ the (pi) rule specializes to four different instances. For instance, take $(s,t) = (\star,\star)$. This means that for $A, B$ such that $\Gamma \vdash A : \star$ and $\Gamma \vdash B : \star$, the product $(\Pi x : A.B)$ can be typed (by $\star$). So, if $\Gamma, x : A \vdash b : B$ then by the (abs) rule we can derive $\Gamma \vdash \lambda x : A.b : (\Pi x : A.B)$. Because $x : A : \star$ and $b : B : \star$, by the definitions in Figure 2 both $x$ and $b$ are terms. So using the rule $(\star,\star)$ we can type terms depending on terms. Using the other rules $(s,t) \in \{\star,\Box\} \times \{\star,\Box\}$ we can type the other dependencies, as described in the table below.

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

The (app) rule is natural given the interpretation of the dependent product ($\Pi$).

The (conv) rule tells us that if we can deduce that $a$ has type $A$, and we can deduce that $A$ is $\beta$-equal to the (typable) expression $B$, that we may deduce that $a$ has type $B$. This rule is necessary because in the $\lambda$-cube reduction is possible on the level of types, so types, like terms, do not have to be in normal form. For instance, suppose that $f = \lambda\alpha : \star.\alpha$ and $id = \lambda\alpha : \star.\lambda x : \alpha.x$, then we have $\vdash id\ (f\ \text{Int}) : (f\ \text{Int}) \to (f\ \text{Int})$. The conversion rule allows us to make the necessary reduction to deduce $\vdash id\ (f\ \text{Int}) : \text{Int} \to \text{Int}$.

## *The $\lambda$-cube and functional programming*

The $\lambda$-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

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

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

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

$$(\text{abs}) \qquad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$$

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

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

$$(\text{conv}) \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash B : s \quad A =_\beta B}{\Gamma \vdash a : B}$$

**Figure 3: Typing rules of the $\lambda$-cube**

force the framework to support or not support each of these features. This flexibility and ability to parameterise makes the $\lambda$-cube a good choice for a basis for a functional programming language.

Unfortunately, despite its flexibility, there are still things that are not possible in the $\lambda$-cube framework as we will illustrate below.

Using the systems in the $\lambda$-cube we can define a number of different identity functions. Using $\lambda{\to}$ we can define the monomorphic identity function on terms (of type $\alpha$) by $(\lambda x : \alpha.x) : \alpha \to \alpha$. Using $\lambda 2$ we can define the polymorphic identity function on terms by $(\lambda\alpha : \star.\lambda x : \alpha.x) : \forall\alpha.\alpha \to \alpha$. Using $\lambda\underline{\omega}$ we can define the monomorphic identity function on types (of kind $\kappa$) by $(\lambda\alpha : \kappa.\alpha) : \kappa \to \kappa$. 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 $\kappa$ as argument and yields the identity function on types of kind $\kappa$. It is tempting to define such a function by: $\lambda\kappa : \Box.\lambda\alpha : \kappa.\alpha : (\Pi\kappa : \Box.\kappa \to \kappa)$. Although this expression is syntacticly valid, it is not possible to type it in one of the type systems of the $\lambda$-cube. To be able to type the expression by using the abstraction rule, we need $\Pi\kappa : \Box.\kappa \to \kappa$ to be typable. The premises of the product rule state that there should be a sort $s$ such that $\vdash \Box : s$ and $(s, \Box) \in R$. Because in the $\lambda$-cube there is no such sort the candidate function cannot be typed. However, if we would extend the sorts with an extra sort, say $\Box'$, add the rule $[\,] \vdash \Box : \Box'$ to the typing rules, and add $(\Box', \Box)$ to the set of rules $R$ we would be able to define a 'polymorphic identity function on types'.

### *Pure Type Systems*

The example above illustrates that the $\lambda$-cube is not general enough for all purposes: the type system described above takes us outside the scope of the $\lambda$-cube. Fortunately, there exists a well studied generalisation of the $\lambda$-cube in which the described system fits. This generalisation is given by the theory of Pure Types Systems (PTSs). The main differences between the $\lambda$-cube and PTSs are that

- in PTSs one can choose the set of sorts $S$ freely, where in the $\lambda$-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 typing rule for the dependent product is generalised in the sense that products need not have the same type as their range. That is, $\Pi x : A.B$ does not necessarily have the same type as $B$. This is reflected in the fact that rules have three components in PTSs instead of two in the $\lambda$-cube: the third component gives the type of the dependent product.
- the set of rules $R$ can be any subset of $S^3$ instead of the sets $\{(\star, \star)\} \subseteq R \subseteq \{\star, \Box\} \times \{\star, \Box\}$. Sometimes we denote rules of the form $(s_1, s_2, s_2) \in S^3$ by $(s_1, s_2) \in S^2$.

$$\begin{array}{lcll}
E & = & V & (\text{variable}) \\
 & | & E\ E & (\text{application}) \\
 & | & \lambda V : E.E & (\text{abstraction}) \\
 & | & \Pi V : E.E & (\text{product}) \\
 & | & S & (\text{sort})
\end{array}$$

**Figure 4: Expressions of PTSs**

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

$$(\text{pi}) \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3} \qquad (s_1, s_2, s_3) \in R$$

**Figure 5: Changed Typing rules of PTSs**

The set of expressions of PTSs is given in Figure 4, the changed typing rules are given in Figure 5.

In the PTS defined by the specification below the 'polymorphic identity function on types' can be typed.

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 $\vdash (\lambda\kappa : \Box.\lambda\alpha : \kappa.\alpha) : (\Pi\kappa : \Box.\kappa \to \kappa)$. The term is typable because we have added an extra layer beyond $\Box$.

## 3. 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. Functional programming languages provide features, such as definitions, algebraic data types and case expressions, which are not in PTSs. To describe these features we need to extend the definitions of the previous chapter.

## Definitions and Recursion

We introduce definitions and recursion into our language by extending the expression grammar with the following production for let expressions:

$$\text{let } V : E = E \text{ in } E$$

Let expressions introduce an extra type of reduction, besides $\beta$-reduction. For instance, inside the body of the expression:

$$\text{let } n : \text{Int} = 5 \text{ in } n + n$$

the variable $n$ reduces to 5. Reduction introduced by a let expression is called $\delta$-reduction. To introduce $\delta$-reduction in the type system of our language we 'store' so-called $\delta$-reduction-rules in the context. In this way a context is a tuple $D;\Gamma$ where $D$ contains the $\delta$-reduction-rules and $\Gamma$ the assumptions.

We change the (conv) rule to incorporate $\delta$-reduction as follows:

$$(\text{conv})\ \frac{D;\Gamma \vdash a : A \quad D;\Gamma \vdash B : s \quad A =_{\beta\delta_D} B}{D;\Gamma \vdash a : B}$$

In this rule $=_{\beta\delta_D}$ stands for the equivalence relation induced by $\beta$-reduction and $\delta$-reduction w.r.t. the rules in $D$.

For instance, using this extended (conv) rule we can derive: $\alpha \rightarrow_\delta \text{Int}; \alpha : \star \vdash (\lambda x : \alpha.x) : \text{Int} \rightarrow \text{Int}$

Finally, we add the following (let) rule to the type system.

$$(\text{let})\ \frac{\begin{array}{c} D, x \rightarrow_\delta a; \Gamma, x : A \vdash a : A \\ D, x \rightarrow_\delta a; \Gamma, x : A \vdash b : B \end{array}}{D;\Gamma \vdash (\text{let } x : A = a \text{ in } b) : B}$$

From now one we will denote a context $D;\Gamma$ by just $\Gamma$ if in a particular rule the set $D$ is not used.

## Algebraic Data Types

Functional programming languages provide means for users to define their own data structures, called algebraic data types.

A simple way to introduce ADTs in a PTS is to treat type- and data constructors as variables. For instance, we can introduce the List data type by adding the following variables to the context.

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

Notice that we have to explicitly provide the types at which data constructors are 'instantiated'. For instance, the empty list of integers is constructed by Nil Int, instead of just Nil as in Haskell.

Defining algebraic data types 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 [12]. We feel this choice would be too restrictive: Haskell (without extensions) only allows 'independent' data constructor arguments. The following data type definition is therefore not allowed: a type constructor E : $\star$ with a single data constructor EC : $\forall a : \star.a \rightarrow (a \rightarrow \text{Int}) \rightarrow \text{E}$. In this definition the type ($a$) of the second argument and the type ($a \rightarrow \text{Int}$) of 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 the example below). Therefore, we have chosen for a more liberal definition than the Haskell definition. Our definition is based on the following requirements, which 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, say $tca_1, \ldots, tca_{\#tca}$, called type constructor arguments. A type constructor applied to the right number of arguments yields an expression of kind type.
- A data constructor $dc_i$ has as first arguments the arguments of its type constructor, and then a number of data constructor arguments, say $dca_1, \ldots, dca_{\#dca_i}$ A data constructor applied to the right number of type- (say $atca_1 \ldots atca_{\#tca}$) and data constructor arguments yields an expression of type $tc\ atca_1 \ldots atca_{\#tca}$.

This leads to the following typing rule for introducing data type definitions.

(We use the following notation: $\overrightarrow{tca} = [tca_1, \ldots, tca_{\#tca}]$, $\overrightarrow{dca}_i = [dca_{i,1}, \ldots, dca_{i,\#dca_i}]$, 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 $\Pi a_1 : A_1 \ldots \Pi a_n : A_n.B$.)

$$\frac{\begin{array}{c} \Gamma \vdash tct : s \\ \forall j.\Gamma, tc : tct \vdash dct_j : t_j \\ \Gamma, tc : tct, dc_1 : dct_1, \ldots, dc_{\#dc} : dct_{\#dc} \vdash a : A \end{array}}{\Gamma \vdash (\text{data } tc : tct \ = \ \{dc_1 : dct_1, \ldots, dc_{\#dc} : dct_{\#dc}\} \text{ in } a) : A}$$

with the side conditions:

- $tct \equiv \Pi\overrightarrow{tca} : \overrightarrow{tcat}.\star$
- $\forall 1 \leq j \leq \#dc : dct_j \equiv \Pi\overrightarrow{tca} : \overrightarrow{tcat}.\Pi\overrightarrow{dca}_j : \overrightarrow{dcat}_j.tc\ \overrightarrow{tca}$

Notice that the usage of the dependent product ($\Pi$) in our definition to 'link' the data constructor arguments makes it possible to let the types of data constructor arguments depend on earlier data constructor arguments. In Haskell this is not possible, because the arrow ($\rightarrow$), which can be seen as an independent product, is used to 'link' the data constructor arguments. For the same reason it is hard to introduce ADTs in PTSs using 'sum' and 'product' types.

## 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.

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 calculates the length of a list can be defined by:

$$\begin{array}{lcl} \text{length} & : & \forall a:\star.\text{List } a \rightarrow \text{Int} \\ \text{length} & = & \lambda a:\star.\lambda xs:\text{List } a.\text{case } xs \text{ of} \\ & & \{\text{Nil } a \quad\quad \Rightarrow \quad 0 \\ & & ;\text{Cons } a\ y\ ys \ \Rightarrow \ 1 + (length\ a\ ys)\} \end{array}$$

The first step in extending the PTS framework with the case construct is to extend our expression grammar with the following production.

$$\text{case } E\ of\ \{[V] \Rightarrow E; \ldots; [V] \Rightarrow E\}$$

The extension of the framework with case expressions induces an extra reduction relation, defined by:

$$\begin{array}{l} \text{case } dc\ \overrightarrow{atca}\ \overrightarrow{adca} \text{ of } \{dc\ \overrightarrow{tca}\ \overrightarrow{dca} \Rightarrow res\} \\ \rightarrow_c \\ res[\overrightarrow{tca} := \overrightarrow{atca}, \overrightarrow{dca} := \overrightarrow{adca}] \end{array}$$

We change the (conv) rule to incorporate case-reduction by changing the side condition to $A =_{\beta\delta c} B$.

The typing rule for case expressions is given below. 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.

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

Here is an example in which the case rule is used:

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

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

The above presented case rule can be used to introduce powerful, and maybe unwanted, dependencies between types and terms. For instance it is possible to define the expression:

$$\begin{array}{ll} (\lambda x : & \text{case } N \text{ of} \\ & \{\text{Zero} \quad \Rightarrow \quad \text{Int} \\ & ;\text{Succ } \_ \ \Rightarrow \quad \text{Bool}\}.x)\ 3 \end{array}$$

This expression is type correct if-and-only-if the term $N$ reduces to Zero. Because it is undecidable whether an expression reduces to a normal form, type checking in a system with the above case rule is undecidable. We can repair this problem by demanding that the right hand sides of a case expression are terms. This can be accomplished by changing the fourth premise of the (case) rule to $\Gamma \vdash t : \star$.

## Existential Data Types

Existential data types, in the form introduced by Läufer and Odersky in [13], come 'for free' in our extended PTSs, whereas in Haskell 'existentials' are introduced as an (rather ad-hoc) extension [11]. In Läufer's construction we consider data constructor arguments of sort type as existentially quantified. For example we say that in the data type: $E : \star, \text{EC} : \forall a : \star.a \rightarrow (a \rightarrow \text{Bool}) \rightarrow \text{E}$, the variable $a$ is existentially quantified. We consider $a$ to be existentially quantified because when we have an expression of type E, say $e$ : E, then we know that there *exists* a type $a$ such that $e = \text{EC}\ a\ x\ f$ with $x : a$ and $f : a \rightarrow \text{Bool}$. 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 with 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.

$$\begin{array}{lcl} Ex & = & [E : \star, \mathrm{EC} : \forall a : \star. a \to (a \to \mathrm{Bool}) \to \mathrm{E}], \\ Nat & = & [\mathrm{Nat} : \star, \mathrm{Zero} : \mathrm{Nat}, \mathrm{Succ} : \mathrm{Nat} \to \mathrm{Nat}] \\ \Gamma & = & Ex \mathbin{+\!\!+} \mathrm{Nat} \end{array}$$

$$\begin{array}{lcl} \mathrm{isZero} & : & \mathrm{Nat} \to \mathrm{Bool} \\ \mathrm{isZero} & = & \lambda n : \mathrm{Nat}. \\ & & \text{case } n \text{ of} \\ & & \{\mathrm{Zero} \quad \Rightarrow \quad \mathrm{True} \\ & & ;\mathrm{Succ}\ m \quad \Rightarrow \quad \mathrm{False} \\ & & \} \end{array}$$

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

then we have:

$$\begin{array}{l} \vdash \mathrm{EC\ Nat\ (Succ\ Zero)\ isZero} : \mathrm{E} \\ \vdash \mathrm{EC\ Bool\ True\ (id\ Bool)} : \mathrm{E} \\ \vdash \mathrm{apply\ (EC\ Nat\ (Succ\ Zero)\ isZero)} : \mathrm{Bool} \\ \vdash \mathrm{apply\ (EC\ Bool\ True\ (id\ Bool))} : \mathrm{Bool} \\ \mathrm{apply\ (EC\ Nat\ (Succ\ Zero)\ isZero)} \twoheadrightarrow_{\beta c} \mathrm{False} \\ \mathrm{apply\ (EC\ Bool\ True\ (id\ Bool))} \twoheadrightarrow_{\beta c} \mathrm{True} \end{array}$$

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

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

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.

## 4. TYPE CHECKING

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 this algorithm can be modified to deal with our extended language.

### Barthe's algorithm

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 $\Gamma$ 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 [21]. Unfortunately, the completeness of these rules is an open problem. For a full analysis of the problems with a proof of the completeness of Pollack's rules see [17].

Barthe's [4] solution to Pollack's problem is to formulate a new (abs) 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.

### *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 small variation on this theme. We say $P$ is injective if it is functional and for every $s_1, s_2, s_2', s_3, s_3' \in S$:

$$\begin{array}{lclcl} (s_1, s_2) \in A & \& & (s_1', s_2) \in A & \Rightarrow & s_1 = s_1' \\ (s_1, s_2, s_3) \in R & \& & (s_1, s_2', s_3) \in R & \Rightarrow & s_2 = s_2' \end{array}$$

In an injective PTS, 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:

For every set $A$, we let $A^{\uparrow}$ denote the set $A \cup \{\uparrow\}$. If $f \in A \to B^{\uparrow}$ and $a \in A$, we write $f\ a\ \downarrow$ to denote $f\ a \neq \uparrow$.

- The map $.^{-} : S^{\uparrow} \to S^{\uparrow}$ is defined by:

$$s^{-} = \begin{cases} s' & \text{if } (s', s) \in A \\ \uparrow & \text{otherwise} \end{cases}$$

- The map $.^{+} : S^{\uparrow} \to S^{\uparrow}$ is defined by:

$$s^{+} = \begin{cases} s' & \text{if } (s, s') \in A \\ \uparrow & \text{otherwise} \end{cases}$$

- The map $\rho : S^{\uparrow} \times S^{\uparrow} \to S^{\uparrow}$ 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 $\mu : S^{\uparrow} \times S^{\uparrow} \to S^{\uparrow}$ is defined by:

$$\mu(s_1, s_2) = \begin{cases} s_3 & \text{if } (s_1, s_3, s_2) \in R \\ \uparrow & \text{otherwise} \end{cases}$$

### *Classification*

Injective PTSs form a class of PTSs for which one can define two 'simple' functions $\mathrm{sort}(.|.), \mathrm{elmt}(.|.) : C \times E^{\uparrow} \to S^{\uparrow}$ (where C denotes the set of contexts) such that:

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

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

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

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

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

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

**Figure 6: Classification Based Rules for PTSs**

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

This result is known as the 'classification lemma', it tells us that for every expression in an injective PTS we can compute the type of the type of the expression by using elmt(.|.). Note that this type of the type of the expression is always a sort. For expressions whose type is a sort we can compute this sort by using sort(.|.).

The definition of the mappings elmt and sort is given in Figure 7. Keeping the typing rules of the PTSs framework, and the definitions of $.^-$, $.^+$, $\rho$, $\mu$ 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 two more complicated rules is the fact that the last two steps in a derivation of the type of a lambda expression are always (abs) and (pi), and the last two steps in a derivation of the type of an application are always (app) and (pi).

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

**Figure 7: Classification Algorithm**

### *Classification Based Rules*

Barthe's *classification based rules* give a sound and complete syntax directed system for the class of injective PTSs. They are based on the classification algorithm, and are given in Figure 6. In the (abs) rules the 'problematic' premise $(\Pi x : A.B) : s$ is changed to a simpler premise involving the functions sort(.|.) and elmt(.|.). Furthermore the (conv) rule is distributed over the other rules using the notion of weak-head reduction.

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

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

Weak-head reduction differs from $\beta$-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

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

The substitution of the 'problematic' premise $(\Pi x : A.B) : s$ in the (abs) rule with premise $\rho(\mathrm{sort}(\Gamma|A), \mathrm{elmt}(\Gamma, x : A|b)) \downarrow$, which is easier to calculate, can be explained as follows: It is easy to see that $(\Pi x : A.B) : s$ together with the other premise $\Gamma, x : A \vdash b : B$ implies $\rho(\mathrm{sort}(\Gamma|A), \mathrm{elmt}(\Gamma, x : A|b)) \downarrow$: 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 $s_1 = \mathrm{sort}(\Gamma|A)$, $s_2 = \mathrm{elmt}(\Gamma, x : A|b)$ and because $(s_1, s_2, s)$ is a rule we have $\rho(\mathrm{sort}(\Gamma|A), \mathrm{elmt}(\Gamma, x : A|b)) \downarrow$.

## Extending Barthe's algorithm to our language

In the previous section we extended the theory of PTSs with definitions, algebraic data types and the case construct. Below 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.

$$\begin{array}{rcl}
\mathrm{sort}(\Gamma|n) & = & \uparrow \\
\mathrm{sort}(\Gamma|\mathrm{Int}) & = & \star \\
\mathrm{sort}(\Gamma|\mathrm{let}\ x : A = a \ \mathrm{in}\ b) & = & \mathrm{sort}(\Gamma, a : A|b) \\
\mathrm{sort}(\Gamma|\mathrm{data}\ tc : tct = \{dc_j : dct_j\}\ \mathrm{in}\ a) & = & \mathrm{sort}(\Gamma, tc : tct, dc_j : dct_j|a) \\
\mathrm{sort}(\Gamma|\mathrm{case}\ e\ \mathrm{of}\ \{dc\ \overrightarrow{tca}\ \overrightarrow{dca} \Rightarrow res\}) & = & \mathrm{sort}(\Gamma, \overrightarrow{tca} : \overrightarrow{tcat}, \overrightarrow{dca} : \overrightarrow{dcat}|res) \\
\\
\mathrm{elmt}(\Gamma|n) & = & \star \\
\mathrm{elmt}(\Gamma|\mathrm{Int}) & = & \Box \\
\mathrm{elmt}(\Gamma|\mathrm{let}\ x : A = a \ \mathrm{in}\ b) & = & \mathrm{elmt}(\Gamma, a : A|b) \\
\mathrm{elmt}(\Gamma|\mathrm{data}\ tc : tct = \{dc_j : dct_j\}\ \mathrm{in}\ a) & = & \mathrm{elmt}(\Gamma, tc : tct, dc_j : dct_j|a) \\
\mathrm{elmt}(\Gamma|\mathrm{case}\ e\ \mathrm{of}\ \{dc\ \overrightarrow{tca}\ \overrightarrow{dca} \Rightarrow res\}) & = & \mathrm{elmt}(\Gamma, \overrightarrow{tca} : \overrightarrow{tcat}, \overrightarrow{dca} : \overrightarrow{dcat}|res)
\end{array}$$

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

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

**Figure 8: Extended Classification Algorithm**

### Extended Classification Algorithm

The functions $\mathrm{sort}(.|.), \mathrm{elmt}(.|.) : C \times E^{\uparrow} \to S^{\uparrow}$, see Figure 7, are extended with the rules in Figure 8

The rules for let, data, $n$ 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.

### Extended Classification Based Rules

To incorporate the notions of case- and $\delta$-reduction into the classification based rules we define the extended weak-head reduction relation, denoted $\rightarrow'_{wh}$:

Extended Weak-head reduction $\rightarrow'_{wh}$ is defined as the smallest relation such that for every $A, B$ and $\overrightarrow{R}$

$$\begin{array}{lcl}
A \rightarrow_{wh} B & \Rightarrow & A \rightarrow'_{wh} B' \\
A\ \overrightarrow{R} \rightarrow_{\delta} B\ \overrightarrow{R} & \Rightarrow & A\ \overrightarrow{R} \rightarrow'_{wh} B\ \overrightarrow{R} \\
A\ \overrightarrow{R} \rightarrow_{c} B\ \overrightarrow{R} & \Rightarrow & A\ \overrightarrow{R} \rightarrow'_{wh} B\ \overrightarrow{R}
\end{array}$$

The extended classification based rules are defined by the original classification based rules (see Figure 6), where $\twoheadrightarrow_{wh}$ is replaced by $\twoheadrightarrow'_{wh}$, and $=_{\beta}$ by $=_{\beta\delta c}$ plus the rules given in Figure 9.

### Undecidability

The extension of PTSs with ADTs, case expressions and definitions causes undecidability of type checking. The extended classification based rules 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) and (let) rules (with side condition $A =_{\beta\delta c} A'$) are 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

$$\text{(let)}\quad \frac{\begin{array}{c} D, x \rightarrow_{\delta} a; \Gamma, x : A \vdash_{cl} a : A' \\ D, x \rightarrow_{\delta} a; \Gamma, x : A \vdash_{cl} b : B \end{array}}{D; \Gamma \vdash_{cl} (\mathrm{let}\ x : A = a\ \mathrm{in}\ b) : B} \quad A =_{\beta\delta c} A'$$

$$\text{(data)}\quad \frac{\begin{array}{c} \Gamma \vdash_{cl} tct :\twoheadrightarrow'_{wh} s \\ \forall j.\Gamma, tc : tct \vdash_{cl} dct_j :\twoheadrightarrow'_{wh} t_j \\ \Gamma, tc : tct, dc_j : dct_j \vdash_{cl} a : A \end{array}}{\Gamma \vdash_{cl} (\mathrm{data}\ tc : tct\ =\ \{dc_j : dct_j\}\ \mathrm{in}\ a) : A}$$

$$\text{(case)}\quad \frac{\begin{array}{c} \Gamma \vdash_{cl} e :\twoheadrightarrow'_{wh} tc\ \overrightarrow{atca} \\ \forall j.\Gamma \vdash_{cl} dc_j\ \overrightarrow{atca} : \Pi\overrightarrow{dca}_j : \overrightarrow{dcat}_j.(tc\ \overrightarrow{atca}) \\ \forall j.\Gamma, \overrightarrow{dca}_j : \overrightarrow{dcat}_j \vdash_{cl} res_j[\overrightarrow{tca} := \overrightarrow{acta}] : t \\ \Gamma \vdash_{cl} t :\twoheadrightarrow'_{wh} s \end{array}}{\Gamma \vdash_{cl} \mathrm{case}\ e\ \mathrm{of}\ \{dc_j\ \overrightarrow{tca}_j\ \overrightarrow{dca}_j \Rightarrow res_j\} : t}$$

$$\text{(int1)}\quad \frac{}{\Gamma \vdash_{cl} n : \mathrm{Int}} \quad n \in \mathbb{N}$$

$$\text{(int2)}\quad \frac{}{\Gamma \vdash_{cl} \mathrm{Int} : \star}$$

**Figure 9: Extended Classification Based Rules**

$\exists N : A \twoheadrightarrow_{\beta\delta c} N,\ A' \twoheadrightarrow_{\beta\delta c} N$. To make sure that types are strongly normalising we should forbid (general) recursion on the level of types and demand that the right hand sides of case-expressions are terms.

If we change the side condition this way, all side conditions of the rules are decidable. Furthermore, because the rules 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.

## 5. OPERATIONAL SEMANTICS

The *operational semantics* of a functional programming language defines how expressions are reduced. Part of the operational semantics of our language is defined by the reduction rules. However, the reduction rules do not specify which subexpression should be reduced when multiple subexpressions of an expression are reducible. Which subexpression of a term should be reduced in such a case is specified by a *reduction strategy.*

A reducible (sub)expression is called a *redex*. We distinguish three kinds of redeces in our language: a (sub)expression M is called a

- a $\beta$-redex, if $M$ is of the form $(\lambda x : A.N)\ M$,
- a case redex, if $M$ is of the form case $dc\ \overrightarrow{M}$ of $\{\ldots\}$ .
- a $\delta$-redex in context $D;\Gamma$, if $M \equiv v$, with $v \rightarrow_\delta e \in D$.

We have chosen the normal order reduction strategy for evaluating expressions. This strategy iteratively reduces the *left-most outer-most* redex.

The left-most outer-most redex of an expression is defined as follows. Let $N \in E$ and let $\overrightarrow{R} \in E$ be all redeces of $N$. We say that $R \in \overrightarrow{R}$ is an outer-most redex, if there exists 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 $N$.

We have chosen for the normal order 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 [16].

We have implemented an interpreter that reduces PTS expressions using the normal order reduction strategy.

## 6. CONCLUSION

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

We have shown that PTSs are at the top of a 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. Dependent on the chosen PTS, our language provides all these features, there is no need for ad-hoc extensions to incorporate them.

Unlike the description of the Henk language in [10] 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 $\lambda$-cube.

Finally, we have presented an efficient type checking algorithm and an interpreter for our language.

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 $\lambda$ or $\Pi$ 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 how much of the source code should be explicitly typed to keep type checking decidable. A possible starting points for such an effort is the work of Barthe and Sørensen [3] on so called *domain free* PTSs in which the domain of $\lambda$-abstractions can be ommited. Another starting point is the work of Pfenning [15] 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.

## 7. REFERENCES

[1] 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.

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

[3] G. Barthe and M.H. Sørensen. Domain-free pure type systems. *Journal of Functional Programming*, 10:417–452, September 2000. Preliminary version in S. Adian and A. Nerode, editors, Proceedings of LFCS'97, LNCS 1234, pp 9-20.

[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] J. Girard. *Interprétation functionelle et élimunation des coupures de l'arithmètique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[8] Ralf Hinze. Polytypic values possess polykinded types. In *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer Verlag, 2000.

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

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

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

[12] Simon Peyton Jones and editors John Hugs. Haskell 98 report. http://www.haskell.org, 1998.

[13] K. Läufer 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.

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

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

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

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

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

[19] G.Huet T. Coquand. The calculus of constructions: a higher order proof system for mechanizing mathematics. Technical report, INRIA, May 1985. no. 401.

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

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