Programming in Martin-Löf's Type Theory

An Introduction

BENGT NORDSTRÖM, KENT PETERSSON
and
JAN M. SMITH

Department of Computer Science
University of Göteborg/Chalmers,
S-41296 Göteborg

CLARENDON PRESS · OXFORD
1990

3

Expressions and definitional equality

This chapter describes a theory of expressions, abbreviations and definitional equality. The theory was developed by Per Martin-Löf and first presented by him at the Brouwer symposium in Holland, 1981; a further developed version of the theory was presented in Siena 1983.

The theory is not limited to type theoretic expressions but is a general theory of expressions in mathematics and computer science. We shall start with an informal introduction of the four different expression forming operations in the theory, then informally introduce arities and conclude with a more formal treatment of the subject.

3.1 Application

In order to see what notions are needed when building up expressions, let us start by analyzing the mathematical expression

$$y + \sin y$$

We can view this expression as being obtained by applying the binary addition operator $+$ on y and $\sin(y)$, where the expression $\sin(y)$ has been obtained by applying the unary function sin on y.

If we use the notation

$$e(e_1, \ldots, e_n)$$

for applying the expression e on $e_1, \ldots, e_n$, the expression above should be written

$$+(y, \sin(y))$$

and we can picture it as a syntax tree:

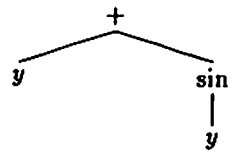

Figure 3.1: Syntax tree for the expression $+(y, \sin(y))$

Similarly, the expression (from ALGOL 68)

```
while x>0 do x:=x-1; f(x) od
```

is analyzed as

```
while(>(x,0),
      ;(:=(x,
           -(x,1)
          ),
       call(f,x)
      )
     )
```

The standard analysis of expressions in Computing Science is to use syntax trees, i.e. to consider expressions being built up from n-ary constants using application. A problem with that approach is the treatment of bound variables.

3.2 Abstraction

In the expression

$$\int_1^x (y + \sin(y))dy$$

the variable y serves only as a placeholder; we could equally well write

$$\int_1^x (u + \sin(u))du \quad \text{or} \quad \int_1^x (z + \sin(z))dz$$

The only purpose of the parts dy, du and dz, respectively, is to show what variable is used as the placeholder. If we let $\square$ denote a place, we could write

$$\int_1^x (\square + \sin(\square))$$

for the expression formed by applying the ternary integration operator $\int$ on the integrand $\square + \sin(\square)$ and the integration limits 1 and x. The integrand has been obtained by functional abstraction of y from $y + \sin(y)$. We will use the notation

$$(x)e$$

for the expression obtained by functional abstraction of the variable x in e, i.e. the expression obtained from e by looking at all free occurrences of the variable x in e as holes. So, the integral should be written

$$\int(((y) + (y, \sin(y))), 1, x)$$

Since we have introduced syntactical operations for both application and abstraction it is possible to express an object by different syntactical forms. An object which syntactically could be expressed by the expression

$$e$$

could equally well be expressed by

$$((x)e)(x)$$

When two expressions are syntactical synonyms, we say that they are *definitionally*, or *intensionally, equal*, and we will use the symbol $\equiv$ for definitional (intensional) equality between expressions. The definitional equality between the expressions above is therefore written:

$$e \equiv ((x)e)(x)$$

Note that definitional equality is a *syntactical* notion and that it has nothing to do with the *meaning* of the syntactical entities.

We conclude with a few other examples of how to analyze common expressions using application and abstraction:

$$\sum_{i=1}^{n} \frac{1}{i^2} \equiv \sum(1, n, ((i)/(1, sqr(i))))$$

$$(\forall x \in \mathsf{N})(x \geq 0) \equiv \forall(\mathsf{N}, ((x) \geq (x, 0)))$$

$$\texttt{for i from 1 to n do } S \equiv for(1, n, ((i)S))$$

3.3 Combination

We have already seen examples of applications where the operator has been applied to more than one expression, for example in the expression $+(y, \sin(y))$.

There are several possibilities to syntactically analyze this situation. It is possible to understand the application operation in such a way that an operator in an application may be applied to any number of arguments. Another way is to see such an application just as a notational shorthand for a repeated use of a binary application operation, that is $e(e_1, \ldots, e_n)$ is just a shorthand for $\ldots((e(e_1))\ldots(e_n))$. A third way, and this is the way we shall follow, is to see the combination of expressions as a separate syntactical operation just as application and abstraction. So if e_1, e_2 ... and e_n are expressions, we may form the expression

$$e_1, e_2, \ldots, e_n$$

which we call the *combination* of e_1, e_2, ... and e_n.

Besides its obvious use in connection with functions of several arguments, the combination operation is also used for forming combined objects such as orderings

$$A, \leq$$

where A is a set and $\leq$ is a reflexive, antisymmetric and transitive relation on A, and finite state machines,

$$S, s_0, \Sigma, \delta$$

where S is a finite set of states, $s_0 \in S$ is an initial state, Σ an alphabet and δ a transition/output function.

3.4 Selection

Given an expression, which is a combination, we can use the syntactical operation *selection* to retrieve its components. If e is a combination with n components, then

$$(e).i$$

is an expression that denotes the i'th component of e if $1 \leq i \leq n$. We have the defining equation

$$(e_1, \ldots, e_n).i \equiv e_i$$

where $1 \leq i \leq n$.

3.5 Combinations with named components

The components of the combinations we have introduced so far have been determined by their *position* in the combination. In many situations it is much more convenient to use names to distinguish the components. We will therefore also introduce a variant where we form a combination not only of expressions but also of names that will identify the components. If e_1, e_2 ... and e_n are expressions and i_1, i_2 ... and i_n, $(n > 1)$, are different names, then we can form the expression

$$i_1 : e_1, i_2 : e_2, \ldots, i_n : e_n$$

which we call a *combination with named components.*

To retrieve a component from a combination with named components, the name of the component, of course, is used instead of the position number. So if e is a combination with names i_1, ..., i_n, then

$$(e).i_j$$

(where i_j is one of $i_1, \ldots, i_n$) is an expression that denotes the component with name i_j.

We will not need combinations with named components in this monograph and will not explore them further.

3.6 Arities

From the examples above, it seems perhaps natural to let expressions in general be built up from variables and primitive constants by means of abstraction, application, combination and selection without any restrictions. This is also the analysis, leaving out combinations, made by Church and Curry and their followers in combinatory logic.

However, there are unnatural consequences of this way of defining expressions. One is that you may apply, e.g., the expression succ, representing the successor function, on a combination with arbitrarily many components and form expressions like $\mathrm{succ}(x_1, x_2, x_3)$, although the successor function only has one argument. You may also select a component from an expression which is not a combination, or select the m'th component $(m > n)$ from a combination with only n components. Another consequence is that self-application is allowed; you may form expressions like succ(succ). Self-application, together with the defining equation for abstraction:

$$((x)d)(e) \equiv d[x := e]$$

where $d[x := e]$ denotes the result of substituting e for all free occurrences of x in d, leads to expressions in which definitions cannot be eliminated. This is seen by the well-known example

$$((x)x(x))((x)x(x)) \equiv ((x)x(x))((x)x(x)) \equiv \ldots$$

From Church [21] we also know that if expressions and definitional equality are analyzed in this way, it will not be decidable whether two expressions are definitionally equal or not. This will have effect on the usage of a formal

system of proof rules since it must be mechanically decidable if a proof rule is properly applied. For instance, in Modus Ponens

$$\frac{A \supset B \qquad A}{B}$$

it would be infeasible to require anything but that the implicand of the first premise is definitionally equal to the second premise. Therefore, definitional equality must be decidable and definitions should be eliminable. The analysis given in combinatory logic of these concepts is thus not acceptable for our purposes. Per Martin-Löf has suggested, by going back to Frege [39], that with each expression there should be associated an *arity*, showing the "functionality" of the expression. Instead of just having one syntactical category of expressions, as in combinatory logic, the expressions are divided into different categories according to which syntactical operations are applicable. The arities are similar to the types in typed λ-calculus, at least from a formal point of view.

An expression is either *combined*, in which case it is possible to select components from it, or it is *single*. Another division is between *unsaturated* expressions, which can be operators in applications, and *saturated* expressions, which cannot. The expressions which are both single and saturated have arity **0**, and neither application nor selection can be performed on such expressions. The unsaturated expressions have arities of the form $(\alpha \twoheadrightarrow \beta)$, where α and β are arities; such expressions may be applied to expressions of arity α and the application gets arity β. For instance, the expression sin has arity $(\mathbf{0} \twoheadrightarrow \mathbf{0})$ and may be applied to a variable x of arity **0** to form the expression $\sin(x)$ of arity **0**. The combined expressions have arities of the form $(\alpha_1 \otimes \ldots \otimes \alpha_n)$, and from expressions of this arity, one may select the i'th component if $1 \leq i \leq n$. The selected component is, of course, of arity α_i. For instance, an ordering $A, \leq$ has arity $(\mathbf{0} \otimes ((\mathbf{0} \otimes \mathbf{0}) \twoheadrightarrow \mathbf{0}))$.

So we make the definition:

Definition 1 *The* arities *are inductively defined as follows*

1. **0** *is an arity; the arity of single, saturated expressions.*
2. *If* $\alpha_1, \ldots, \alpha_n$ $(n \geq 2)$ *are arities, then* $(\alpha_1 \otimes \cdots \otimes \alpha_n)$ *is an arity; the arity of a combined expression.*
3. *If* α *and* β *are arities, then* $(\alpha \twoheadrightarrow \beta)$ *is an arity; the arity of unsaturated expressions.*

The inductive clauses generate different arities; two arities are equal only if they are syntactically identical. The arities will often be written without parentheses; in case of conflict, like in

$$\mathbf{0} \twoheadrightarrow \mathbf{0} \otimes \mathbf{0}$$

$\twoheadrightarrow$ will have lower priority than $\otimes$. The arity above should therefore be understood as

$$(\mathbf{0} \twoheadrightarrow (\mathbf{0} \otimes \mathbf{0}))$$

We always assume that every variable and every primitive (predefined) constant has a unique arity associated with it.

The arities of some of the variables and constants we have used above are:

Expression	Arity
y	**0**
x	**0**
1	**0**
sin	$\mathbf{0} \twoheadrightarrow \mathbf{0}$
succ	$\mathbf{0} \twoheadrightarrow \mathbf{0}$
$+$	$\mathbf{0} \otimes \mathbf{0} \twoheadrightarrow \mathbf{0}$
$\int$	$((\mathbf{0} \twoheadrightarrow \mathbf{0}) \otimes \mathbf{0} \otimes \mathbf{0}) \twoheadrightarrow \mathbf{0}$

From the rules of forming expressions of a certain arity, which we will give, it is easy to derive the arities

Expression	Arity
$\sin(y)$	**0**
$+(y, \sin(y))$	**0**
$(y) + (y, \sin(y))$	$\mathbf{0} \twoheadrightarrow \mathbf{0}$
$\int((y) + (y, \sin(y)), 1, x)$	**0**
$\mathrm{succ}(x)$	**0**

However, neither succ(succ) nor $\mathrm{succ}(x)(x)$ can be formed, since succ can only be applied to expressions of arity **0** and $\mathrm{succ}(x)$ is a complete expression which can not be applied to any expression whatsoever.

3.7 Definitions

We allow abbreviatory definitions (macros) of the form

$$c \equiv e$$

where c is a unique identifier and e is an expression without free variables. We will often write

$$c(x_1, x_2, \ldots, x_n) \equiv e$$

instead of

$$c \equiv (x_1, x_2, \ldots, x_n)e$$

In a definition, the left hand side is called *definiendum* and the right hand side *definiens*.

3.8 Definition of what an expression of a certain arity is

In the rest of this chapter, we will explain how expressions are built up from variables and primitive constants, each with an arity, and explain when two expressions are (definitionally, intensionally) equal.

1. *Variables.* If x is a variable of arity α, then

   $$x$$

   is an expression of arity α.

2. *Primitive constants.* If c is a primitive constant of arity α, then

   $$c$$

   is an expression of arity α.

3. *Defined constants.* If, in an abbreviatory definition, the definiens is an expression of arity α, then so is the definiendum.

4. *Application.* If d is an expression of arity $\alpha \twoheadrightarrow \beta$ and a is an expression of arity α, then

   $$d(a)$$

   is an expression of arity β.

5. *Abstraction.* If b is an expression of arity β and x a variable of arity α, then

   $$((x)b)$$

   is an expression of arity $\alpha \twoheadrightarrow \beta$. In cases where no ambiguities can occur, we will remove the outermost parenthesis.

6. *Combination.* If a_1 is an expression of arity α_1, a_2 is an expression of arity α_2, ... and a_n is an expression of arity α_n, $2 \leq n$, then

   $$a_1, a_2, \ldots, a_n$$

   is an expression of arity $\alpha_1 \otimes \alpha_2 \otimes \cdots \otimes \alpha_n$.

7. *Selection.* If a is an expression of arity $\alpha_1 \otimes \cdots \otimes \alpha_n$ and $1 \leq i \leq n$, then

   $$(a).i$$

   is an expression of arity α_i.

3.9 Definition of equality between two expressions

We will use the notation $a : \alpha$ for a is an expression of arity α and $a \equiv b : \alpha$ for a and b are equal expressions of arity α.

1. *Variables.* If x is a variable of arity α, then

   $$x \equiv x : \alpha$$

2. *Constants.* If c is a constant of arity α, then

   $$c \equiv c : \alpha$$

3. *Definiendum ≡ Definiens.* If a is a definiendum with definiens b of arity α, then

   $$a \equiv b : \alpha$$

4. *Application 1.* If $a \equiv a' : \alpha \twoheadrightarrow \beta$ and $b \equiv b' : \alpha$, then

   $$a(b) \equiv a'(b') : \beta$$

5. *Application 2. (β-rule).* If x is a variable of arity α, a an expression of arity α and b an expression of arity β, then

   $$((x)b)(a) \equiv b[x := a] : \beta$$

   provided that no free variables in a becomes bound in $b[x := a]$.

6. *Abstraction 1. (ξ-rule).* If x is a variable of arity α and $b \equiv b' : \beta$, then

   $$(x)b \equiv (x)b' : \alpha \twoheadrightarrow \beta$$

7. *Abstraction 2. (α-rule).* If x and y are variables of arity α and $b : \beta$, then

   $$(x)b \equiv (y)(b[x := y]) : \alpha \twoheadrightarrow \beta$$

   provided that y does not occur free in b.

8. *Abstraction 3. (η-rule).* If x is a variable of arity α and b is an expression of arity $\alpha \twoheadrightarrow \beta$, then

   $$(x)(b(x)) \equiv b : \alpha \twoheadrightarrow \beta$$

   provided that x does not occur free in b.

9. *Combination 1.* If $a_1 \equiv a_1' : \alpha_1$, $a_2 \equiv a_2' : \alpha_2$, ... and $a_n \equiv a_n' : \alpha_n$, then

   $$a_1, a_2, \ldots, a_n \equiv a_1', a_2', \ldots, a_n' : \alpha_1 \otimes \alpha_2 \otimes \cdots \otimes \alpha_n$$

10. *Combination 2.* If $e : \alpha_1 \otimes \cdots \otimes \alpha_n$ then

$$(e).1, (e).2, \ldots, (e).n \equiv e : \alpha_1 \otimes \cdots \otimes \alpha_n$$

11. *Selection 1.* If $a \equiv a' : \alpha_1 \otimes \cdots \otimes \alpha_n$ and $1 \leq i \leq n$, then

$$(a).i \equiv (a').i : \alpha_i$$

12. *Selection 2.* If $a_1 : \alpha_1, \ldots, a_n : \alpha_n$ and $1 \leq i \leq n$ then

$$(a_1, \ldots a_n).i \equiv a_i : \alpha_i$$

13. *Reflexivity.* If $a : \alpha$, then $a \equiv a : \alpha$.

14. *Symmetry.* If $a \equiv b : \alpha$, then $b \equiv a : \alpha$.

15. *Transitivity.* If $a \equiv b : \alpha$ and $b \equiv c : \alpha$, then $a \equiv c : \alpha$.

From a formal point of view, this is similar to typed λ-calculus. The proof of the decidability of equality in typed λ-calculus can be modified to yield a proof of decidability of $\equiv$. It is also possible to define a normal form such that an expression on normal form does not contain any subexpressions of the forms $((x)b)(a)$ and $(a_1, \ldots, a_n).i$. It is then possible to prove that every expression is definitionally equal to an expression on normal form. Such a normalization theorem, leaving out combinations, is proved in Bjerner [14].

A note on the concrete syntax used in this book

When we are writing expressions in type theory we are not going to restrict ourselves to prefix constants but will use a more liberal syntax. We will freely use parentheses for grouping and will in general introduce new syntax by explicit definitions, like

$$(\Pi x \in A)B(x) \equiv \Pi(A, B)$$

If x is a variable of arity $\alpha_1 \otimes \cdots \otimes \alpha_n$ we will often use a form of pattern matching and write

$$(x_1, \ldots, x_n)e$$

instead of $(x)e$ and, correspondingly, write x_i instead of $x.i$ for occurrences of $x.i$ in the expression e.

Part I

Polymorphic sets