Programming Language Concepts — The Lambda Calculus Approach[*]

Maarten M. Fokkinga
Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

1987

The Lambda Calculus is a formal system, originally intended as a tool in the foundation of mathematics, but mainly used to study the concepts of algorithm and effective computability. Recently, the Lambda Calculus and related systems acquire attention from Computer Science for another reason too: several important programming language concepts can be explained elegantly and can be studied successfully in the framework of the Lambda Calculi. We show this mainly by means of examples. We address ourselves to interested computer scientists who have no prior knowledge of the Lambda Calculus. The concepts discussed include: parameterization, definitions, recursion, elementary and composite data types, typing, abstract types, control of visibility and life-time, and modules.

1. Introduction

The Lambda Calculus is a completely formally defined system, consisting of *expressions* (for functions or rather algorithms) and *rules* that prescribe how to evaluate the expressions. It has been devised in the thirties by Alonzo Church to study the concept of function (as a *recipe*) and to use it in the foundation of mathematics. This latter goal has not been achieved (although recent versions of the Lambda Calculus come quite close to it, see Martin-Löf [16], Coquand & Huet [6]); the former goal, the study of the concept of function, has led to significant contributions to the theory of effective computability.

Recently, the Lambda Calculus and related systems, together called Lambda Calculi, have aroused much interest from computer science because several important programming language concepts are present — or can be expressed faithfully — in them in the most pure form and without restrictions that are sometimes imposed in commercial programming languages. Expressing a programming language concept in the Lambda Calculus has the following benefits:

- It may shed some light upon the concept and thus give some insight.
- It may answer fundamental questions about the concept via theorems already available in the Lambda Calculus. And there are quite a lot of them.

[*]Originally appeared in:
Essays on Concepts, Formalisms, and Tools (edited by P.R.J. Asveld and A. Nijholt), CWI Tract 42, 1987, pp. 129–162. ISNB 90-6196-326-5

- It proves that the concept does not add something essentially new to the language. In particular it guarantees that there can not be nasty interferences with other such concepts. (It has always been a problem in programming language design to combine concepts that are useful in isolation but have unexpected and undesired interferences when taken together.)

Some programming language concepts cannot be expressed in the Lambda Calculus. In view of the expressive power of the Lambda Calculus one should first become suspicious of such a concept. Secondly, one may try to extend or adapt the Lambda Calculus in such a way that the concept is expressible. The techniques and tools developed for the Lambda Calculus may then prove useful to study the extension. Examples of such concepts are *assignment* and *exception handling.*

In this paper our aim is to show the significance of the Lambda Calculus approach to "Programming Language Concepts", and to raise interest in the Lambda Calculi. We address ourselves to (experienced) programmers; no knowledge of the Lambda Calculus is assumed. To this end we keep the formalism to a bare minimum and use computer science terms and notations as much as possible. We discuss a variety of programming language concepts, such as parameterization, definition, recursion, elementary and composite data types, typing, abstract types, control of visibility and life-time, and modules. All this is preceded by a brief exposition of the Lambda Calculus and its role as an area of research in itself.

The importance of the Lambda Calculus for the design of programming languages has already been recognized in the sixties by Landin [12, 13, 14]. Algol 68's orthogonality is very similar to the simplicity of the Lambda Calculus. Reynolds [26] explains the essence of Algol as follows:

> "Algol is obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus."

2. The Lambda Calculus

We describe the Lambda Calculus as a mini programming language in a notation and a terminology that is conventional in computer science. (Thus the title of this section might have read: "Lambda Calculus Concepts — the programming language approach"). Some topics of past and current research are mentioned.

Expressions. We assume that some set of *identifiers* is available, and we let x denote an arbitrary identifier. For *expressions* there are three syntactic formation rules:

e	$::=$	x	identifier
e	$::=$	$(\mathbf{fn}\, x \bullet e)$	function expression
e	$::=$	$e(e)$	function call

We let e, ef, ea, $eb, \ldots$ denote arbitrary expressions; (f is mnemonic for function, a for argument and b for body). Fully capitalized *WORDS* will abbreviate specific expressions. The sign $\equiv$ stands for syntactic equality.

Notes
2.1. In an expression $(\mathbf{fn}\, x \bullet eb)$, x is called *its parameter*, eb *its body*; the parameter is a local name, so that renaming of the parameter and all its occurrences in the body is allowed and is considered not to change the expression. We thus identify $(\mathbf{fn}\, x \bullet x)$ and $(\mathbf{fn}\, y \bullet y)$, both denoting the identity function as we shall see below.

2.2. We leave out parentheses if no confusion can result; this is often the case with the outermost parentheses of a function expression under the convention that the expression following the dot $\bullet$ should be taken as large as possible.

2.3. An expression $\mathbf{fn}\, x \bullet eb$ is an anonymous function. In contrast to conventional languages, the concepts of function and of naming are separated here syntactically. Naming is discussed in Section 3.1.

2.4. Church originally wrote $\hat{x}\, e$ for $\mathbf{fn}\, x \bullet e$, but for typographical reasons changed this to $\wedge\, x\; e$ and later to $\lambda\, x\; e$. This is still the standard notation and clearly explains the part 'Lambda' in the name.

2.5. As an example, consider the expression $\mathbf{fn}\, x \bullet f(f(x))$ which we shall call TWICE_f: called with argument x_0 this function will return $f(f(x_0))$, i.e., the result of calling f twice. The function $\mathbf{fn}\, f \bullet \mathit{TWICE}_f$ will return for each argument function f the function that calls f twice. So both the argument and the result of a function may be functions themselves. Actually, each expression denotes a function.

We shall now formally define the semantics of expressions analogously to the way in primary school children are taught to evaluate fractions like $(5 \times 8 + 8)/(10 \times 8)$: they are given some simplification rules that may be applied in any order.

Evaluation. An expression e *evaluates to* an expression e', notation $e \Rightarrow e'$, if e' is obtained from e by repeatedly (zero or more times) applying the following *evaluation* rule:

$$\text{replace a part} \quad (\mathbf{fn}\, x \bullet eb)(ea) \quad \text{by} \quad [ea/x]eb.$$

Here, and in the sequel, $[ea/x]eb$ denotes the result of substituting ea for each occurrence of x in eb (taking care to avoid clash of names by renaming local identifiers where appropriate).

Notes
2.6. Substitution is a syntactic manipulation that is tedious to define formally. Let it suffice here to say that $[ea/f]\; f(f(\mathbf{fn}\, x \bullet x))$ equals $ea(ea(\mathbf{fn}\, x \bullet x))$, and that

$$[...x.../f]\;\; f(\mathbf{fn}\, x \bullet f(x)) \;\;\equiv\;\; (...x...)(\mathbf{fn}\, x' \bullet\; (...x...)(x')),$$

where x' is a new identifier distinct from x.

2.7. As an example we have:

$$\begin{array}{ll} & (\mathbf{fn}\, f \bullet \mathit{TWICE}_f)(\mathit{sin})(\mathit{zero}) \\ \Rightarrow & \\ & \mathit{TWICE}_{\mathit{sin}}(\mathit{zero}) \qquad \text{i.e., } (\mathbf{fn}\, x \bullet \mathit{sin}(\mathit{sin}(x)))(\mathit{zero}) \\ \Rightarrow & \\ & \mathit{sin}(\mathit{sin}(\mathit{zero})) \end{array}$$

and this cannot be evaluated further at this point.

2.8. In Algol 60 jargon [23] the evaluation rule is the *body replacement rule*: the effect of a function call is explained by replacing it by the function body with substitution of the argument for the parameter. In the Lambda Calculus the rule is called the β-rule, and evaluation is called *reduction*.

This seemingly simple mini programming language gives rise to a large number of thorough questions that in turn have led to substantial research efforts and a lot of results. We mention but a few.

1. Do there exist expressions whose evaluation may not terminate? Answer: yes there are, for we shall see that arbitrary recursive definitions are expressible.

2. Is the evaluation strategy (the choice what part to evaluate next) of any importance? Answer: different strategies cannot yield different final outcomes, but one may terminate in cases where the other does not. Also the number of evaluation steps to reach the final outcome, if any, depends on the strategy.

3. Is it possible to express numbers and to do arithmetic in the Lambda Calculus? Answer: yes, see Section 3.4.

4. Clearly, function expressions denote functions in the sense of *recipes* of how to obtain the result when given an argument. Is it possible to interpret expressions as functions in the sense of *a set of (argument, result)-pairs*, such a set itself being a possible argument or result? Answer: this has been a long standing problem. D. Scott formulated the first such models in 1969; a lot of others have been found since.

5. When may or must expressions be called semantically *equivalent*? (Of course we want semantic equivalence to satisfy the usual laws of equality and to be preserved under evaluation.) If two expressions may both evaluate to a common intermediate or final outcome, they must be called equivalent. However, it is possible that they do so only "in the limit", after an infinite number of evaluation steps; in this case they may be called equivalent. And what about calling expressions equivalent if they have no outcome, not even in the limit?

6. What are the consequences of the restriction that in $\mathbf{fn}\, x \bullet eb$ parameter x must occur at least once in the body eb? And of the extra evaluation rule

   $$\text{replace} \quad \mathbf{fn}\, x \bullet ef(x) \quad \text{by} \quad ef$$

   (because both denote the same function intuitively)?

More information about the Lambda Calculus may be obtained from [1, 11, 29].

3. Basic Programming Language Concepts

In this section we express various basic programming language concepts in the Lambda Calculus. Justified by this, we also give specific syntactic forms for each concept together with *derived* evaluation rules.

3.1. Definitions. We extend the syntactic formation rules for expressions by:

$$e \quad ::= \quad (\mathbf{df}\, x = e \bullet e) \quad \text{definition expression}$$

Within $(\mathbf{df}\, x = ea \bullet eb)$ the part $x = ea$ is a *local definition* that extends over the body eb. We consider this new expression as an abbreviation for $(\mathbf{fn}\, x \bullet eb)(ea)$, so that the Lambda Calculus is not extended in an essential way, and the evaluation rule has to read:

$$\text{replace a part} \quad (\mathbf{df}\, x = ea \bullet eb) \quad \text{by} \quad [ea/x]eb.$$

Notes

3.1.1. The definition $x = ea$ in $\mathbf{df}\, x = ea \bullet eb$ is nonrecursive. This is a consequence of our choice to let it abbreviate $(\mathbf{fn}\, x \bullet eb)(ea)$.

3.1.2. By construction there is a close correspodence, or rather identity, between defitions $\mathbf{df}\, x = ea \bullet eb$ and parameterizations as in $(\mathbf{fn}\, x \bullet eb)(ea)$. This can be taken as a guiding principle in the design of programming languages:

> for each kind of parameter (think of **value**, **in**, **out**, **ref** and **name**) there exists a semantically identical definition, and conversely.

The consequences of adhering to this *Principle of Correspondence* have been worked out by Tennent [30]. Pascal strongly violates it.

3.1.3. There is another principle involved here, the *Principle of Naming.* In $\mathbf{df}\, x = ea \bullet eb$ identifier x names ea locally in eb, and both ea, eb and x are arbitrary. This principle is violated in Pascal, because e.g. statements cannot be named and naming can be done only locally to procedure and function bodies.

3.1.4. We have explained the local name introduction of $\mathbf{df}\, x = ea \bullet eb$ in terms of the $\mathbf{fn}$-construct. Reynolds [26] makes this to a guiding principle for the design of programming languages:

> every local name introduction can be explained by the $\mathbf{fn}$-construct.

We shall apply this *Principle of Locality* to the expression for recursion, below.

3.1.5. As an example, we now name the function *TWICE*:

$$\mathbf{df}\, \mathit{twice} = (\mathbf{fn}\, f \bullet (\mathbf{fn}\, x \bullet f(f(x)))) \bullet \; \ldots \mathit{twice}(\mathit{sin})(\mathit{zero}) \ldots.$$

Here *sin* and *zero* are just identifiers for which a definition may be provided in the context; (Principle of Naming).

3.2. Multiple Parameters and Definitions. Consider once again the expression $\mathit{TWICE}(f_0)(x_0)$, where $\mathit{TWICE} \equiv \mathbf{fn}\, f \bullet (\mathbf{fn}\, x \bullet f(f(x)))$. One may easily verify that $\mathit{TWICE}(f_0)(x_0) \Rightarrow (\mathbf{fn}\, x \bullet f_0(f_0(x)))(x_0) \Rightarrow f_0(f_0(x_0))$. We might say that *both f and x* are parameters, and *both f_0 and x_0* are arguments. Thus multiple parameters are possible, for which we design a special syntax:

$$e \quad ::= \quad (\mathbf{fn}\, x_1, \ldots, x_n \bullet e) \qquad \text{for distinct } x_1, \ldots, x_n$$

$$e \quad ::= \quad ef(e, \ldots, e)$$

These expressions are to abbreviate $(\mathbf{fn}\, x_1 \bullet (\ldots(\mathbf{fn}\, x_n \bullet e)\ldots))$ respectively $ef(e_1)\ldots(e_n)$, so that the evaluation rule has to read:

$$\text{replace} \quad (\mathbf{fn}\, x_1, \ldots, x_n \bullet e)(e_1, \ldots, e_n) \quad \text{by} \quad [e_1, \ldots, e_n/x_1, \ldots, x_n]e.$$

Guided by the Principle of Correspondence we also design the corresponding definition form:

$$e \quad ::= \quad (\mathbf{df}\, x_1 = e_1, \ldots, x_n = e_n \bullet e) \qquad \text{for distinct } x_1, \ldots, x_n$$

with evaluation rule:

$$\text{replace} \quad \mathbf{df}\, x_1 = e_1,\ \ldots,\ x_n = e_n \bullet e \quad \text{by} \quad [e_1, \ldots, e_n/x_1, \ldots, x_n]e.$$

Notes
3.2.1. For example, we may now write $TWICE'(f_0, x_0)$ where $TWICE' \equiv \mathbf{fn}\, f, x \bullet f(f(x))$. We can also write $\mathbf{df}\, f = f_0,\ x = x_0 \bullet f(f(x))$.

3.2.2. The industrious reader may verify that the multiple definitions and substitutions are *simultaneous* rather than sequential: it turns out that the definition $x_i = e_i$ extends only over e and not over e_1 through en. The distinctness of $x_1, \ldots, x_n$ is necessary to formulate the evaluation rule so simple (and to guarantee that the substitution is well defined).

3.2.3. Exercise. Let $\mathbf{df}\, x_1 = e_1;\ \ldots;\ x_n = e_n \bullet e$ abbreviate the *sequential* definition $(\mathbf{df}\, x_1 = e_1 \bullet (\ldots(\mathbf{df}\, x_n = e_n \bullet e)\ldots))$. Now think about the corresponding "sequential parameterization".

3.3. Recursion. A recursive definition is a definition in which the defined name occurs in the defining expression. A stupid evaluation strategy that first of all tries to eliminate the recursively defined name will therefore certainly get into an infinite loop of evaluation steps. However, sometimes (unfortunately not always) the recursively defined name occurs in a subexpression (*then*- or *else*-branch in particular) whose evaluation is not needed to reach the final outcome. A moderately clever evaluation strategy will not attempt to evaluate such needless occurrences. Thus the concept of recursion is fully captured by an expression in which designated occurrences evaluate —if time has come— to the expression itself. We "extend" (not really, see Note 3.3.4 below) the Lambda Calculus by the following grammar and evaluation rules:

$$e \quad ::= \quad (\mathbf{rec}\, x \bullet e) \qquad \text{recursion expression}$$
$$\text{replace} \quad (\mathbf{rec}\, x \bullet e) \quad \text{by} \quad [(\mathbf{rec}\, x \bullet e)/x]e$$

Notes
3.3.1. Within $(\mathbf{rec}\, x \bullet e)$ the occurrences of x in e are the points of recursion: such an occurrence evaluates to the original recursive expression. (But if such an occurrence is contained in a *then*- or *else*-branch, it may happen that after one expansion it is not any more subject to the above evaluation rule.)

3.3.2. The concepts of recursion and of definition have been separated syntactically. We may combine them by abbreviating $(\mathbf{df}\, x = (\mathbf{rec}\, x \bullet ea) \bullet eb)$ by $(\mathbf{df}\,\mathbf{rec}\, x = ea \bullet eb)$: the occurrences of x in eb as well as in ea will evaluate —if time has come— to the recursive expression $(\mathbf{rec}\, x \bullet ea)$.

3.3.3. Assuming that *if then else* and arithmetic are possible, we may write the definition of the factorial function as follows:

$$\begin{array}{ll} & \mathbf{df}\,\mathbf{rec}\, \mathit{fac} = (\mathbf{fn}\, n \bullet \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n \times \mathit{fac}(n-1)) \\ \text{i.e.,} & \mathbf{df}\, \mathit{fac} = (\mathbf{rec}\, \mathit{fac} \bullet (\mathbf{fn}\, n \bullet \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n \times \mathit{fac}(n-1))) \\ \equiv & \mathbf{df}\, \mathit{fac} = (\mathbf{rec}\, f \bullet (\mathbf{fn}\, n \bullet \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n \times f(n-1))) \end{array}$$

The part $(\mathbf{rec}\, f \bullet \ldots f(n-1))$ denotes the factorial function without giving it a name that can be used elsewhere.

3.3.4. In $(\mathbf{rec}\, x \bullet e)$ the identifier x is a local name whose scope extends over e. Following the Principle of Locality we explain that local naming in terms of the **fn**-construct: provided that *REC* satisfies the property

$$\mathit{REC}\,(\mathbf{fn}\, x \bullet e) \quad \text{evaluates to} \quad [\mathit{REC}\,(\mathbf{fn}\, x \bullet e)/x]e$$

we may consider $(\mathbf{rec}\, x \bullet e)$ to abbreviate $\mathit{REC}\,(\mathbf{fn}\, x \bullet e)$. We could now add a constant '*REC*' to the Lambda Calculus with the above evaluation rule, but it turns out (space limitations prohibit to give the motivation) that we may take:

$$\mathit{REC} \equiv \mathbf{fn}\, f \bullet W_f(W_f) \quad \text{where} \quad W_f \equiv \mathbf{fn}\, y \bullet f(y(y))$$

as is easily verified.

3.3.5. Notice that recursion (as in the **rec**-expression), circularity (as in the **df rec**-definition), self-activation (as in the evaluation rule for **rec**) and self-application (as in W_f: y is applied to itself) are intimately related.

3.3.6. Mutual recursion can also be expressed, but we shall not do so here.

3.4. Truth Values and Enumerated Types. We shall choose two expressions *TRUE* and *FALSE* and some function expressions *AND*, *OR*, *NOT* and *IF* such that the laws that we expect to hold, are indeed true of these expressions. The observable behaviour of *TRUE* and *FALSE* is in their being used as the condition part of an *IF* call: we wish to have

$$\begin{array}{lll} \mathit{IF}(\mathit{TRUE}, e_1, e_2) & \Rightarrow & e_1, \\ \mathit{IF}(\mathit{FALSE}, e_1, e_2) & \Rightarrow & e_2. \end{array}$$

Hence we let *TRUE* and *FALSE* be selector functions:

$$\begin{array}{lll} \mathit{TRUE} & \equiv & \mathbf{fn}\, x, y \bullet x \\ \mathit{FALSE} & \equiv & \mathbf{fn}\, x, y \bullet y \end{array}$$

so that we may choose

$$\mathit{IF} \equiv \mathbf{fn}\, b, x, y \bullet b(x, y).$$

The evaluation property for IF is true indeed. Now functions AND, OR and NOT are easy to define:

$$\begin{aligned} AND &\equiv \textbf{fn}\, b1, b2 \bullet IF(b1, b2, FALSE), \\ OR &\equiv \textbf{fn}\, b1, b2 \bullet IF(b1, TRUE, b2), \\ NOT &\equiv \textbf{fn}\, b \bullet IF(b, FALSE, TRUE). \end{aligned}$$

Notes

3.4.1. Plugging in the expression IF into the expression NOT and performing some evaluation steps, we see that we also may set $NOT \equiv \textbf{fn}\, b \bullet b(FALSE, TRUE)$. Similarly for AND and OR.

3.4.2. Suppose $PROG$ is a program (an expression) in which identifiers *True*, *False*, *If*, *And*, *Or* and *Not* occur and have been assumed to satisfy the usual Boolean laws. We may then form

$$\textbf{df}\ True = TRUE,\ False = FALSE,\ \ldots,\ Not = NOT \bullet PROG.$$

In other words, the definitions $True = TRUE, \ldots, Not = NOT$ can be considered to belong to the standard environment and the application programmer need not know the particular representation choices made for truth values. We shall see in Section 4 how to hide the representation choices so that the application programmer is not allowed to write $True(e_1, e_2)$ (but has to use the *If*-function explicitly).

3.4.3. Rather than providing a standard environment we can also design specific syntactic expressions for truth values, thus:

$$\begin{aligned} e &::= \textbf{true} \mid \textbf{false} \mid (e\, \textbf{and}\, e) \mid (e\, \textbf{or}\, e) \mid (\textbf{not}\, e) \\ e &::= \textbf{if}\, e\, \textbf{then}\, e\, \textbf{else}\, e \end{aligned}$$

together with the *derived* evaluation rules:

replace (**true and** e) by e
replace (**false and** e) by e
replace **if true then** e_1 **else** e_2 by e_2
$\vdots$

3.4.4. Elements of a finite enumeration type can be represented analogously: selector function $\textbf{fn}\, x_1, \ldots, x_n \bullet x_i$ represents the ith element, and $elt(e_1, \ldots, e_n)$ is the implementation of

case *elt* **in** 1: e_1, ..., n: e_n **endcase**.

3.4.5. The above representation has been chosen in the assumption that the evaluation strategy does not evaluate the argument expressions before the body replacement rule is applied. Otherwise both the **then**- and the **else**-branch are always evaluated, and that is undesirable. We can, however, adapt the representation to that strategy, but we shall not discuss it here.

3.5. Arithmetic: (Natural) Numbers. Throughout the paper we say 'number' instead of 'natural number' (0,1,2,...). As motivated below in Note 3.5.3 we choose to represent number n by an n-fold repeated call of a function f on an initial argument a, where both f and a are parameters:

$$\mathbf{fn}\, f, a \bullet f(\ldots(f(f(a)))\ldots) \quad (n \text{ times an } f)$$

In particular we set

$$\begin{array}{lll} \mathit{ZERO} & \equiv & \mathbf{fn}\, f, a \bullet a \\ \mathit{ONE} & \equiv & \mathbf{fn}\, f, a \bullet f(a) \\ \mathit{TWO} & \equiv & \mathbf{fn}\, f, a \bullet f(f(a)). \end{array}$$

The successor function SUCC may be implemented by

$$\begin{array}{lll} \mathit{SUCC} & \equiv & \mathbf{fn}\, n \bullet \text{ ``}n\text{+1-fold iteration''} \\ & \equiv & \mathbf{fn}\, n \bullet \ (\mathbf{fn}\, f, a \bullet f(\text{``}n\text{-fold iteration of } f \text{ on } a\text{''})) \\ & \equiv & \mathbf{fn}\, n \bullet \ (\mathbf{fn}\, f, a \bullet f(n(f, a))) \end{array}$$

For example, one easily verifies that

$$\mathit{SUCC}(\mathit{TWO}) \Rightarrow \mathbf{fn}\, f, a \bullet f(\mathit{TWO}(f, a)) \Rightarrow \mathbf{fn}\, f, a \bullet f(f(f(a)))$$

which represents 3. The test for equality is also easy:

$$\mathit{EQ0} \equiv \mathbf{fn}\, n \bullet n(F, \mathit{TRUE}) \qquad \text{where } F \equiv \mathbf{fn}\, x \bullet \mathit{FALSE}$$

so that

$$\begin{array}{l} \mathit{EQ0}(\mathit{ZERO}) \Rightarrow \mathit{ZERO}(F, \mathit{TRUE}) \Rightarrow \mathit{TRUE}, \\ \mathit{EQ0}(\mathit{ONE}) \Rightarrow \mathit{ONE}(F, \mathit{TRUE}) \Rightarrow F(\mathit{TRUE}) \Rightarrow \mathit{FALSE}. \end{array}$$

The construction of a predecessor function is more complicated. The idea is to reconstruct the number itself, n say, and simultaneously "remember" at each step in the reconstruction the outcome of the previous step. So, each intermediate result consists of a pair, in which one component is a number (initially 0 and at most n) and the other component its predecessor. We use here pair-expressions of the form $\langle e_1, e_2 \rangle$ and suffixes .1 and .2 for selection of the first and second component of a pair; in Section 3.6 we show how to express these in the Lambda Calculus. Now we set

$$\mathit{PRED} \equiv \mathbf{fn}\, n \bullet \mathit{FINISH}(n(F, A))$$

where

$$\begin{array}{lll} A & \equiv & \langle \mathit{ZERO}, \mathit{DONTCARE} \rangle \\ F & \equiv & \mathbf{fn}\, \mathit{pair} \bullet \langle \mathit{SUCC}(\mathit{pair}.1), \mathit{pair}.2 \rangle \\ \mathit{FINISH} & \equiv & \mathbf{fn}\, \mathit{pair} \bullet \mathit{pair}.2 \end{array}$$

Notes

3.5.1. The choice for $\mathit{DONTCARE}$ in PRED determines the outcome of $\mathit{PRED}(\mathit{ZERO})$. If no outcome is wanted, because within the set of numbers zero has no predecessor, we may take a nonterminating expression like $(\mathbf{rec}\, x \bullet x)$.

3.5.2. One way to define addition is:

$$\mathbf{df\ rec}\ \mathit{add}\ =\ \mathbf{fn}\, m, n \bullet \mathit{IF}(\mathit{EQ}0(m), n, \mathit{add}(\mathit{PRED}(m), \mathit{SUCC}(n)))$$

However, use of recursion expressions is not necessary:

$$\mathbf{df}\ \mathit{add}\ =\ \mathbf{fn}\, m, n \bullet m(\mathit{SUCC}, n).$$

One can prove that all effectively computable total functions on numbers can be defined solely in terms of the number zero, the successor function and so-called primitive recursions (of higher order). For our representation it turns out that we can express primitive recursion of f and a as: $\mathbf{fn}\, n \bullet n(f, a)$. So let $\mathit{NREC} \equiv \mathbf{fn}\, f, a \bullet (\mathbf{fn}\, n \bullet n(f, a))$. Then knowledge of the representation of numbers is not needed any more, and in particular we can replace all expressions '$n(\mathit{ef}, \mathit{ea})$' above by '$\mathit{NREC}(\mathit{ef}, \mathit{ea})(n)$'.

3.5.3. The representation choice might be motivated thus: we have represented the data structure "number" by its most characteristic control structure, namely the use of it to control a repetition (or: repeated call; compare $\mathbf{var}\, x := a$; $\mathbf{for}\, i := 1\, \mathbf{to}\, n\, \mathbf{do}\, x := f(x)$ with $f(\ldots(f(f(a)))\ldots)$). A better motivation reads as follows. Numbers form an inductively definable data type: Zero is a "number" and if n is a "number" then so is Succ(n). If we are able to replace Zero and Succ in an arbitrary "number" Succ(... (Succ(Succ(Zero)))...) by any a and f, then we are effectively able to construct all functions on "numbers" that are definable by (structural) induction. Thus Succ(...(Succ(Succ(Zero)))...) is represented by

$$\mathbf{fn}\, \mathit{Succ}, \mathit{Zero} \bullet \mathit{Succ}(\ldots(\mathit{Succ}(\mathit{Succ}(\mathit{Zero})))\ldots)\ \equiv\ \mathbf{fn}\, f, a \bullet f(\ldots(f(f(a)))\ldots),$$

and primitive recursion has been built in.

3.5.4. One might object to the above representation of numbers: it can hardly be called a faithful modeling of commercial programming languages, because the representation length, and therefore storage space too, for a number n is linear in n and also the number of evaluation steps to compute the predecessor of m, or the sum of m and n, is linear in m. We can however improve upon this drastically. Observe that the above representation is close to the unary notation of numbers: number 1...11 (in unary notation) has been represented by $\mathbf{fn}\, f, a \bullet f(\ldots(f(f(a))\ldots)$. Now we represent e.g. number 1001101 (in binary notation) by $\mathbf{fn}\, f, g, a \bullet f(g(g(f(f(g(f(a)))))))$; an f for 1 and a g for 0. The representation length grows only logarithmically. The successor function may be defined thus:

$$\mathit{SUCC}'\ \equiv\ \mathbf{fn}\, n \bullet \mathbf{fn}\, f, g, a \bullet \mathit{FINISH}(n(F, G, \langle \mathit{CARRY}, a\rangle))$$

where

$$\begin{array}{lll}
\mathit{CARRY} & \equiv & \mathbf{fn}\, x, y \bullet x\ \ (\equiv\ \mathit{TRUE}) \\
\mathit{NOCARRY} & \equiv & \mathbf{fn}\, x, y \bullet y\ \ (\equiv\ \mathit{FALSE}) \\
F & \equiv & \mathbf{fn}\, \langle \mathit{carry}, \mathit{result}\rangle \bullet \mathit{carry}(g, f)(\mathit{result}) \\
G & \equiv & \mathbf{fn}\, \langle \mathit{carry}, \mathit{result}\rangle \bullet \mathit{carry}(f, g)(\mathit{result}) \\
\mathit{FINISH} & \equiv & \mathbf{fn}\, \langle \mathit{carry}, \mathit{result}\rangle \bullet \mathit{carry}(f(\mathit{result}), \mathit{result})
\end{array}$$

In a similar way addition can be defined. It turns out that evaluation of both $SUCC(n)$ and $PRED(n)$ takes O(log n) steps, and $ADD(m, n)$ takes O(log m + log n) steps. No programming language can improve upon this whenever it allows unbounded numbers.

3.5.5. Similar remarks as in Notes 3.4.2-3 apply here as well. In view of the complicated representation of numbers, and implementation of the operations, this is very welcome.

3.6. Composite Data Types: Records and Lists. We can be very brief with respect to lists. Note 3.5.3 provides the clue to choose the representation: the list

$$\text{Cons } (x_1, \text{ Cons } (x_2, \ldots \text{Cons } (x_n, \text{ Nil})\ldots))$$

is represented by

$$\begin{array}{l} \mathbf{fn}\, \mathit{Cons}, \mathit{Nil} \bullet \mathit{Cons}(x_1, \mathit{Cons}(x_2, \ldots \mathit{Cons}(x_n, \mathit{Nil}) \ldots)) \qquad \equiv \\ \mathbf{fn}\, f,\, a \bullet f(x_1, f(x_2, \ldots f(x_n, a) \ldots)). \end{array}$$

We leave it to the reader to define functions *NIL*, *CONS* and *LREC* (cf. *ZERO*, *SUCC* and *NREC* of Section 3.5), and to build *HEAD*, *TAIL*, *EQNIL* and so on in terms of them. (Typed versions will be given in Section 5.3).

In the next section we discuss typing and shall require that lists be homogeneous: all elements of a list must belong to the same data type. So we need a kind of **record**-construct for inhomogeneous aggregates. For simplicity we discuss pairs (2-tuples) only; the generalization to n-tuples is straightforward. The tuple Pair(x,y) is represented by $\mathbf{fn}\, \mathit{Pair} \bullet \mathit{Pair}(x, y)$, i.e. $\mathbf{fn}\, f \bullet f(x, y)$. The constituting elements can be retrieved by applying the pair to the appropriate selector functions. Thus we let

$$\begin{array}{lcl} \langle e_1, e_2 \rangle & \equiv & \mathbf{fn}\, f \bullet f(e_1, e_2) \\ e.1 & \equiv & e(\mathbf{fn}\, x, y \bullet x) \\ e.2 & \equiv & e(\mathbf{fn}\, x, y \bullet y) \end{array}$$

or we introduce the left-hand sides as new syntactic forms, together with the appropriate, derived, evaluation rules:

$$\begin{array}{l} e \ ::= \ \langle e, e \rangle \mid e.1 \mid e.2 \\ \text{replace } \ \langle e_1, e_2 \rangle.1 \ \text{ by } \ e_1 \\ \text{replace } \ \langle e_1, e_2 \rangle.2 \ \text{ by } \ e_2. \end{array}$$

3.7. Concluding Remarks.
3.7.1. We have shown how data structures may be represented by functions. Reynolds [25] and Meertens [17] show the usefulness of such representations in practice. (However, they term the technique *procedural data abstraction* rather than procedural ($\approx$ functional) data representation.)

3.7.2. In a similar way arbitrary Turing machines and similar devices can be represented by functions, see Fokkinga [7] and Langmaack [15]. It turns out that the functions do accept functions as parameters, but do not yield functions as result: the representation can therefore be carried out in conventional languages (if recursive types are available, as in Algol 68). From this, one immediately concludes several fundamental limitations of compile-time checks.

4. Typing

We consider *typing* a well-formedness check where attributes, called *types*, are assigned to subexpressions and the type of a subexpression has to satisfy specific requirements in relation to the types of its direct constituent parts. An expression that passes the check is said to be *typed* or *typable*.

The assignment of types to expressions may be facilitated by an explicitly written type at each introduction of a local name; but this is not necessary. In the former case we speak of *explicit* typing, in the latter case of *implicit* typing or type deduction. An expression that can be assigned only one type is called *monomorphic*. An expression is called *polymorphic* if it is assigned many related types, a type scheme so to speak. In particular, a function expression is polymorphic if it may be applied to arguments of various but schematically the same types. We call a function *generic* if it may be applied to a type (which may determine the types of the following arguments and final result). (Another term for genericity is *parametric polymorphism.*) Examples will be given in the sequel. The type of a function whose arguments must have type **nat** and whose result has type **bool**, is written (**nat** → **bool**).

In this section we discuss the Monomorphic Typing M, the Polymorphic Typing P and the Generic Typing G. These are extensively studied by Hindley & Seldin [11]. Other overviews on typing are given by Reynolds [28] and Cardelli & Wegner [4]; they cover more features than we do.

4.1. The Usefulness of Typing. Typing proves its usefulness if the typable expressions satisfy a useful semantic property (chosen by the designer of the typing). We list here some properties that may or may not be aimed at in the design of a typing.

1. Set theoretic interpretation. For the class of typable expressions a simple set-theoretic interpretation is possible, in which expressions of type (**nat** → **bool**) are interpreted as *mappings* from the set of numbers to the set of truth values, rather than *recipes* that prescribe how to obtain the outcome when given an argument. (This property precludes self-application and therefore also the unrestricted use of the **rec**-expression.)

2. Termination. The evaluation of typable expressions terminates. One might argue that non-terminating evaluations are useless, but apart from that, the existence of nonterminating expressions invalidates conventional mathematical laws such as

$$0 \times e = 0 \qquad \text{for any expression } e \text{ of type } \mathbf{nat}.$$

(This property too precludes general recursion.)

3. Implementation ease. For typable expressions the size of the storage space for the values that appear during the evaluation, is compile-time computable. This property is aimed at by the Pascal typing; consequently the programmer is forced to specify the size of arrays by constants. The property eases the task of the implementor, not of the programmer.

4. Representation independence. The outcome of typable expressions does not depend on the representation chosen for internally used data like truth values, numbers and other data types. This property allows the implementor to switch freely from the unary representation to the binary representation; cf. Section 3.5 and Note 3.5.4. Moreover, the implementor may even implement arithmetic in hardware: the outcome of typable expressions will not change.

This property, as well as property 1 precludes the use of **nat**-expressions as functions even if we know they are; cf. Section 3.5.

5. Error prevention. For typable expressions many errors of the kind "Ah, of course, I see, this is a misprint" and "Ah, of course, this is an oversight" are impossible. This is a rather fuzzy property and much of it is implied by properties 1 and 4.

4.2. The Monomorphic Typing M. We describe here a simple typing M that gives the essence of Pascal-like typing. We concentrate on the Lambda Calculus and shall *derive* the M-typing requirements for the derived expressions.

M-types. The attributes assigned to expressions, and called M-*types*, are syntactic forms defined by the following grammar:

$$t ::= (t \to t) \mid \mathbf{nat} \mid \mathbf{bool} \mid \mathbf{char} \mid \ldots .$$

We let t, ta, tb denote arbitrary types.

M-typable expressions. We write the type assigned to a (sub)expression as a superscript. It is required that within $(\mathbf{fn}\, x \bullet e)$ all occurrences of x in e have the same type; this type is written at the parameter position: $(\mathbf{fn}\, x^t \bullet e)$. Now consider the following infinite set of grammar rules, one for each choice of t, ta, and tb:

$$\begin{array}{lll} e^{t} & ::= & x^{t} \\ e^{ta\to tb} & ::= & (\mathbf{fn}\, x^{ta} \bullet e^{tb}) \\ e^{tb} & ::= & e^{ta\to tb}(e^{ta}) \end{array}$$

The grammar generates, by definition, the M-typed expressions. Alternatively we may consider it as a formalization of the requirements for a M-type assignment:

- if x has been assigned type ta and e type tb, then $(\mathbf{fn}\, x \bullet e)$ may be assigned type $(ta \to tb)$;
- if ef has been assigned type $ta \to tb$ and ea type ta, then $ef(ea)$ may be assigned type tb;
- if identifier x has been assigned type t, then considered as a subexpression it may be assigned type t.

For example, for any t the expression

$$((\mathbf{fn}\, x^{tt} \bullet x^{tt})^{tt\to tt}(\mathbf{fn}\, x^{t} \bullet x^{t})^{t\to t})^{tt}$$

where $tt \equiv t \to t$, is M-typed. But

$$(\mathbf{fn}\, id \bullet id(id))(\mathbf{fn}\, x \bullet x)$$

is not M-typable although it evaluates in one step to the preceding expression.

Notes

4.2.1. One may succeed easily in deriving the M-typing requirements for derived expressions like $\textbf{fn}\, x, y \bullet e$ and $\textbf{df}\, x \;=\; ea \bullet eb$. Extend the grammar for types by

$$t \quad ::= \quad (t_1, \ldots, t_n \to t)$$

where $t_1, \ldots, t_n \to t$ is thought of as an abbreviation of $t_1 \to (t_2 \to \ldots (t_n \to t) \ldots)$. Then the grammar for typed expressions may be extended by

$$\begin{array}{lll} e^{t_1,\ldots,t_n \to t} & ::= & (\textbf{fn}\, x_1^{t_1}, \ldots, x_n^{t_n} \bullet e^t) \\ e^t & ::= & e^{t_1,\ldots,t_n \to t}(e^{t_1}, e^{t_2}, \ldots, e^{t_n}) \\ e^t & ::= & (\textbf{df}\, x^{ta} = e^{ta} \bullet e^t). \end{array}$$

4.2.2. We may decide to assign *ZERO*, *ONE*, *TWO*... type **nat**, and *SUCC* type **nat** $\to$ **nat**; and so on:

$$\begin{array}{lll} e^{\textbf{nat}} & ::= & \mathit{ZERO} \mid \mathit{ONE} \mid \mathit{TWO} \mid \ldots \\ e^{\textbf{nat} \to \textbf{nat}} & ::= & \mathit{SUCC} \\ e^{\textbf{bool}} & ::= & \mathit{TRUE} \mid \mathit{FALSE} \\ e^{\textbf{bool},\textbf{bool} \to \textbf{bool}} & ::= & \mathit{AND} \mid \mathit{OR} \\ e^{\textbf{bool},t,t \to t} & ::= & \mathit{IF}. \end{array}$$

A justification for this decision is given in Section 5.2. Notice that different occurrences of *IF* may be assigned different types; *IF* is a polymorphic expression. However, in

$$\textbf{df}\, \mathit{If} = \mathit{IF} \bullet \ldots . \mathit{If} \ldots . \mathit{If} \ldots . \mathit{If} \ldots .$$

there is only one occurrence of *IF*, so that all occurrences of *If* are assigned the same type **bool**, T, $T \to T$ for one specific type T. In this way the programmer is forced to spell out *IF* each time again. One solution to this problem is given in Section 4.3: polymorphic typing. Another solution is to extend the grammar by:

$$e^t \quad ::= \quad \mathit{if}\ e^{\textbf{bool}}\ \mathit{then}\ e^t\ \mathit{else}\ e^t.$$

This is justified by considering $\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2$ as an abbreviation of $\mathit{IF}(e, e_1, e_2)$. Yet another solution is given in Section 4.4: generic typing.

4.2.3. We may treat the other polymorphic expressions similarly to *IF*: build the polymorphism into the derived syntax and typing. For example for pairs:

$$\begin{array}{llll} t & ::= & \langle t, t \rangle & \text{type for pairs} \\ e^{\langle t_1, t_2 \rangle} & ::= & \langle e^{t_1}, e^{t_2} \rangle & \\ e^t & ::= & e^{\langle t, t' \rangle}.1 & \\ e^t & ::= & e^{\langle t', t \rangle}.2 & \end{array}$$

and analogously for n-tuples, lists, arrays and so on.

4.2.4. The M-typing as described so far validates properties 1, 2, 4 and 5 of Section 4.1.

4.2.5. Expression *REC* is not M-typable. So the following typing rule properly extends the set of M-typable expressions:

$$e^t \quad ::= \quad (\mathbf{rec}\, x^t \bullet e^t)$$

or equivalently; cf. Note 4.2.1.

$$e^{(t\to t)\to t} \quad ::= REC.$$

Now property 2 (Termination) is invalidated, the other three are preserved.

4.2.6. One may extend the monomorphic typing by allowing recursive types, as in Algol 68. It turns out that every expression of the pure Lambda Calculus is typable, with the recursive type $fun = fun \to fun$. Nevertheless not all expressions are typable, and properties 4 and 5 remain valid, if we require that *ZERO* is assigned type **nat** only, and *SUCC* type **nat** → **nat** and so on. We shall not discuss recursive types any further.

4.3. Polymorphic Typing P. The monomorphic typing M has a flagrant deficiency: there are only monomorphic types and consequently one is forced to duplicate expressions solely for the purpose of letting different occurrences be assigned different types. For example consider

$\mathbf{df}\; id \;=\; (\mathbf{fn}\, x \bullet x)$
  $\bullet \ldots id(zero^{\mathbf{nat}}) \ldots id(true^{\mathbf{bool}}) \ldots id(id) \ldots$
$\mathbf{df}\; compose \;=\; (\mathbf{fn}\, f, g \bullet (\mathbf{fn}\, x \bullet f(g(x))))$
  $\bullet \ldots compose(not^{\mathbf{bool}\to\mathbf{bool}}, not^{\mathbf{bool}\to\mathbf{bool}}) \ldots$
   $\ldots compose(sqr^{\mathbf{nat}\to\mathbf{nat}}, ord^{\mathbf{char}\to\mathbf{nat}}) \ldots$
$\mathbf{df}\; sort$ = sorting function
  $\bullet \ldots sort$(number list) $\ldots sort$(character list) …

These expressions are not M-typable, but after substituting the defining expressions for the defined identifiers (or multiplicating the definitions, one for each use) they are M-typable. The solution to this deficiency is simple: polymorphism. It has been introduced into computer science by Milner [20], but was already known in the Lambda Calculus as Principal Typing.

P-types. We let z range over identifiers. (One may stipulate that the identifiers denoted by z are distinct from those denoted by x, but this is not necessary). The P-types are now defined thus:

$$t \quad ::= \quad z \mid (t \to t) \mid \mathbf{nat} \mid \mathbf{bool} \mid \mathbf{char} \mid \ldots.$$

The identifiers occurring in P-types shall play the role of place holders for which arbitrary types may be substituted consistently. An P-type may therefore be considered as an "M-type *scheme*".

P-typable expressions. There is only one difference in the type assignment rules in comparison with those of the M-typing:

> within $\mathbf{df}\; x = ea^t \bullet eb$ the occurrences of x in eb may be assigned *instantiations* of t, more precisely: P-types must be substituted for the identifiers in t that have been used in the typing of ea^t only (and not in its context); different occurrences of x may be assigned different instantiations of t.

All the other rules for the M-typing are valid for the P-typing as well.

Notes
4.3.1. The examples above are all P-typable. For instance:

$$\begin{array}{l} \mathbf{df}\ id\ =\ (\mathbf{fn}\, x^a \bullet x^a)^{a\to a} \\ \quad \bullet\ \ldots\ id^{\mathbf{nat}\to\mathbf{nat}}(zero^{\mathbf{nat}})\ \ldots \\ \quad\quad \ldots\ id^{\mathbf{bool}\to\mathbf{bool}}(true^{\mathbf{bool}})\ \ldots \\ \quad\quad \ldots\ id^{(b\to b)\to(b\to b)}(id^{(b\to b)})\ \ldots \end{array}$$

But unfortunately, $(\mathbf{fn}\, id \bullet \ldots id(zero^{\mathbf{nat}}) \ldots id(true^{\mathbf{bool}}) \ldots)(\mathbf{fn}\, x \bullet x)$ is not P-typable. This also shows that the P-typing violates the Principle of Correspondence.

4.3.2. The polymorphic typing is used in modern functional languages, like Miranda [31, 32], as well as in the modern imperative language ABC [18].

4.3.3. One obvious advantage of the P-typing over the more “powerful” G-typing of the next subsection, is that types need not be written explicitly in the program text (although it is permitted): the type-checker will deduce them anyway (and show or insert them on request).

4.3.4. The language can be enriched by further constructs for the definition of user defined types. One particularly simple and elegant way has been built in in Miranda [31, 32]. We discuss type definitions more fundamentally in the next subsection.

4.4. Generic Typing G. The P-typing, although quite successful for programming in the small, is not very satisfactory for at least two reasons. First, as we have seen in Note 4.3.1 parameters can not be used polymorphically. Second, the facility of assigning **nat** to *ZERO*, *ONE*, ... and **nat** $\to$ **nat** to *SUCC* is not generally available to the programmer. (Recall that *ZERO*, *ONE*, ... *SUCC* are merely ordinary expressions.) The programmer does need such a facility in order to get Representation Independence for his own devised data types. The solution is to control the type assignment explicitly, by indicating for each parameter the desired type and in addition allowing types to be parameters (“genericity”). The resulting language is often called Second Order Lambda Calculus and was invented by J.-Y Girard and, independently, Reynolds [24].

G-types. As before z ranges over identifiers. G-types are defined by the following grammar.

$$t \ ::= \ z \mid (t \to t) \mid (z : \mathbf{tp} \to t).$$

The third form of type is called a *generic type.* Within $(z : \mathbf{tp} \to t)$ identifier z is a local name whose scope extends over t; of course systematic renaming is allowed. We let t, ta, tb,... denote arbitrary types. (Type constants like **nat** and **bool** are no longer needed. We shall see that the programmer can “define” them.)

G-Typable expressions. As for the M-typing we define:

$$\begin{array}{lll} e^t & ::= & x^t \\ e^{ta\to tb} & ::= & (\mathbf{fn}\, x^{ta} : ta \bullet e^{tb}) \qquad \text{Note the explicit type for } x \\ e^{tb} & ::= & e^{ta\to tb}(e^{ta}) \end{array}$$

Henceforth we shall omit a type superscript at a parameter, if it also occurs explicitly. We add two new expressions:

$e^{z:\mathbf{tp}\to t}$	$::=$	$(\mathbf{fn}\, z : \mathbf{tp} \bullet e^t)$	generic function expression
$e^{[ta/z]t}$	$::=$	$e^{z:\mathbf{tp}\to t}(ta)$	generic instantiation/call

Deliberately, generic instantiation looks like a normal function call, but it is not: it is a new kind of expression with a *type* as one of its direct constituents. Similarly for generic function expression. Within $(\mathbf{fn}\, z : \mathbf{tp} \bullet e)$ identifier z is a local name whose scope extends over e; z may e.g. occur in the explicit types in e. For simple examples see the first note below; Section 5 contains further examples.

Evaluation. For the new expressions we have to define an evaluation rule. The rule is evident:

replace $(\mathbf{fn}\, z : \mathbf{tp} \bullet e)(ta)$ by $[ta/z]e$.

Notes
4.4.1. For example, the generic identity function reads $GID \equiv \mathbf{fn}\, z : \mathbf{tp} \bullet (\mathbf{fn}\, x : z \bullet x^z)^{z\to z}$ and has type $z : \mathbf{tp} \to (z \to z)$. The generic instantiation $GID(nat)$ has type $[nat/z](z \to z) \equiv nat \to nat$ and evaluates to $(\mathbf{fn}\, x : nat \bullet x^{nat})^{nat\to nat}$ as expected and desired. Similarly, $GID(bool)$ has type $[bool/z](z \to z) \equiv bool \to bool$, and evaluates to the identity function for *bool*-expressions.

4.4.2. In Section 5 we shall introduce some syntactic sugar like we did before. Let it suffice here that we may write the left-hand sides for the right-hand sides:

$(z : \mathbf{tp}, x : z \to z)$	$(z : \mathbf{tp} \to (x : z \to z))$
$(\mathbf{fn}\, z : \mathbf{tp}, x : z \bullet z)$ $(\equiv GID')$	$\mathbf{fn}\, z : \mathbf{tp} \bullet (\mathbf{fn}\, x : z \bullet x)$ $(\equiv GID)$
$GID'(nat, zero^{nat})$	$GID(nat)(zero^{nat})$
and	
$(\mathbf{df}\, z : \mathbf{tp} = nat,\ x : z = zero^{nat} \bullet x)$	$GID'(nat, zero^{nat})$.

4.4.3. The richness of the G-typable expressions is already perceptible from the possibility of generically calling GID with its own type: $GID(z : \mathbf{tp} \to (z \to z))$ has type $(z : \mathbf{tp} \to (z \to z)) \to (z : \mathbf{tp} \to (z \to z))$ and evaluates to the identity function $\mathbf{fn}\, x : (z : \mathbf{tp} \to (z \to z)) \bullet x$. This in turn may be applied to GID and then evaluates to GID again.

4.4.4. The following theorems have been proved for the class of G-typed expressions. For (a)-(e) see [9] and for (f) see [27].

a. Different evaluation strategies cannot produce different outcomes.
b. Any evaluation of any expression terminates.
c. G-typability is preserved under evaluation.
d. The G-type of an expression is uniquely determined and compile-time computable.
e. The generic function expressions and instantiations are semantically insignificant. That is, they can be eliminated compile-time from any expression (by compile-time evaluation), provided, of course, that neither the expression nor the global identifiers in it have a generic part $(z : \mathbf{tp} \to \ldots)$ in their type. For example $GID(nat)$ has type

$nat \to nat$; it evaluates compile-time to $\mathbf{fn}\, x : nat \bullet x$ that contains no generic constructs any more. *GID* itself does contain a generic construct that can not be eliminated, for its type explicitly demands so.

f. A classical set-theoretic interpretation of expressions and types is not possible.

4.4.5. It is still a topic for research how much of the explicit types and generic functions and instantiations can be left out of expressions, while still keeping G-typability decidable.

4.4.6. It is easy to extend the language with a recursive construct; this however invalidates the Termination property (b) above.

4.4.7. A technical detail. Consider $(\mathbf{fn}\, z : \mathbf{tp} \bullet e^t)^{z:\mathbf{tp}\to t}$ and assume that some global identifier x with type $\ldots z \ldots$ occurs in it. There are now a global z and a local z involved in the type assignment to e. To avoid problems one should either forbid such occurrences of x or else require that $[z'/z]e$ has type $[z'/z]t$ for some brand-new identifier z', rather than that e has type t).

5. Type Definitions, Abstract Types and Modules

Clearly a typing is not satisfactory if there is no facility for something like "user defined types", "abstract types" and "modules". Pascal, Algol 68, Ada, Modula 2 and others all have their own way to do so, and the result is an astonishing diversity of different constructs; (think only of type definitions and the problem of choosing between occurrence equivalence, name equivalence and structural equivalence). We have refrained from designing such facilities in an ad-hoc way, because we get them for free, in a fundamental way, from the G-typing: according to the Principle of Correspondence we may write a generic instantiation of a generic function expression as a definition: a type definition. This is done in Section 5.1; Section 5.2 and 5.3 give some examples and Section 5.4 discusses modules. Throughout Section 5.1-3 we use the G-typed Lambda Calculus.

5.1. User Defined Types. Like we did for the un-, M- and P-typed Lambda Calculus, we introduce some special syntactic forms for special (frequently used) composite expressions. The abbreviations for (normal) multiple parameters, arguments and definitions are straightforward; both with respect to the form of the expressions, as well as with respect to the typing and evaluation rules. But generic types, functions and instantiations call for an abbreviation too; in particular we write the left-hand sides, below, for the right-hand sides:

$z : \mathbf{tp},\, t_1,\, \ldots,\, t_n \to t$	$z : \mathbf{tp} \to (t_1 \to (\ldots(t_n \to t)\ldots))$
$\mathbf{fn}\, z : \mathbf{tp},\, x_1 : t_1, \ldots,\, x_n : t_n \bullet e$	$\mathbf{fn}\, z : \mathbf{tp} \bullet (\mathbf{fn}\, x_1 : t_1 \bullet (\ldots(\mathbf{fn}\, x_n : t_n \bullet e)\ldots))$
$ef(t, e_1, \ldots, e_n)$	$ef(t)(e_1)\ldots(e_n)$
$\mathbf{df}\, z : \mathbf{tp} = t,\, x_1 : t_1 = e_1, \ldots,\, x_n : t_n = e_n \bullet e$	$(\mathbf{fn}\, z : \mathbf{tp},\, x_1 : t_1, \ldots,\, x_n : t_n \bullet e)(t, e_1, \ldots, e_n)$

Notice that in all these expressions the scope of z extends over $t_1, \ldots, t_n$ and e but not over $e_1, \ldots, e_n$. This is particularly true of the **df**-expression. Hence the *derived* typing rules have to read

$$\begin{array}{lll}
e^{z:\mathbf{tp},t_1,\ldots,t_n\to t} & ::= & (\mathbf{fn}\, z : \mathbf{tp},\, x_1 : t_1, \ldots,\, x_n : t_n \bullet e^t) \\
e^{[t/z]tb} & ::= & e^{z:\mathbf{tp},t_1,\ldots,t_n\to tb}(t,\, e_1^{[t/z]t_1}, \ldots,\, e_n^{[t/z]t_n}) \\
e^{[t/z]tb} & ::= & (\mathbf{df}\, z : \mathbf{tp} = t,\, x_1 : t_1 = e_1^{[t/z]t_1}, \ldots,\, x_n : t_n = e_n^{[t/z]t_n} \bullet e^{tb})
\end{array}$$

In words: if z is defined to be t and an expression e_i outside the scope of $z : \mathbf{tp}$ is required to have — formulated inside the scope of z — type t_i, then e_i must actually have $[t/z]t_i$, i.e. the required type t_i in which t is read for z. Within the scope of a type definition $z = t$ identifier z is a type that is unrelated to t as far as type-checking is concerned. (Thus far all our type-checking rules require exact matching, i.e. equality; there has not been introduced any notion of type equivalence.)

Notes
5.1.1. Referring to the expressions discussed above, the collection

$$z : \mathbf{tp},\ x_1 : t_1, \ldots,\ x_n : t_n$$

constitutes the signature of an abstract data type, z being the name for the carrier. The collection

$$t,\ e_1, \ldots,\ e_n$$

constitutes the/an implementation; t being the representation type, i.e. the type to represent the "abstract z-values", and $e_1, \ldots, e_n$ being the implementation of $x_1 : t_1, \ldots, x_n : t_n$. The G-typed Lambda Calculus provides no way to express laws between the $x_1, \ldots, x_n$ that one might wish to hold. See also Section 6.

5.1.2. One might introduce a new expression

$$\mathbf{df}\, z \equiv t \bullet e \quad \text{to stand for} \quad [t/z]e.$$

In this expression, z and t may be used interchangeably within e: as is to be seen in the right-hand side all z's are replaced by t. So here the definition $z \equiv t$ is completely transparent for e. We shall not use this construct in the sequel.

5.2. Simple Abstract Types: ***nat*****.** We have already seen how numbers may be represented by function expressions and that the definitions for zero, successor and primitive recursion in principle suffice to define the other total functions on numbers. We shall now adapt the expressions to the G-typing and provide suggestive names (identifiers) for them.

Remember, number n was represented by

$$\mathbf{fn}\, f, a \bullet f(\ldots(f(f(a))\ldots).$$

This may be typed

$$(\mathbf{fn}\, f : (t \to t),\ a : t \bullet\ f(\ldots(f(f(a))\ldots)))^{(t \to t), t \to t}$$

for any type t. Therefore we make t to an explicit parameter (called z), getting

$$\mathbf{fn}\, z : \mathbf{tp},\ f : (z \to z),\ a : z \bullet f(\ldots(f(f(a))\ldots).$$

The type of these expressions is abbreviated NAT, so

$$NAT \ \equiv\ (z : \mathbf{tp},\ (z \to z),\ z \ \to z).$$

Now we form, given a user program $PROG$:

$$\begin{aligned}
&\mathbf{df}\ \ nat : \mathbf{tp} \ = \ NAT, \\
&\quad zero : nat \ = \ (\mathbf{fn}\, z : \mathbf{tp},\ f : (z \to z),\ a : z \bullet a), \\
&\quad succ : nat \to nat \\
&\qquad = \ \mathbf{fn}\, n : NAT \bullet \ \mathbf{fn}\, z : \mathbf{tp},\ f : z \to z,\ a : z \bullet f(n(z, f, a)), \\
&\quad nrec : \ (z : \mathbf{tp},\ (z \to z),\ z \to (nat \to z)) \\
&\qquad = \ \mathbf{fn}\, z : \mathbf{tp},\ f : z \to z,\ a : z \bullet (\mathbf{fn}\, n : NAT \bullet n(z, f, a)) \\
&\bullet \ \ PROG
\end{aligned}$$

Notes

5.2.1. Notice that uses of number representations have to get an explicit type argument, which determines (see NAT) the types of the subsequent arguments and the final result.

5.2.2. Within $PROG$ the identifiers can only be used as prescribed by their type at the left-hand sides of the definitions; other use within $PROG$ is not G-typable. In particular, although *zero* evaluates to a function, the "expression" $zero\,(bool,\ (\mathbf{fn}\, x : bool \bullet x),\ true^{bool})$ is type-incorrect.

5.2.3. The right-hand sides may be replaced by G-typed versions of the "binary" representation suggested in Note 3.5.4: the entire **df**-expression remains G-typable. Also, as long as $PROG$ has no *nat* in its type, the outcome does not change by this replacement. Cf. 4.1.4.

5.2.4. Nothing prevents us from replacing a by $f(f(f(a)))$ in the right-hand side of the definition of *zero*: what results is G-typable again, but does not have the intended semantics.

5.2.5. Suppose $PROG$ has type *nat*. Then the entire expression has type NAT (not *nat*). Hence the context of the entire expression "knows" that the outcome is a generic iterator function, rather than a number, and it may use the outcome accordingly. This is not at all surprising if one realizes that it is the very context writer who also provides the right-hand sides (and possibly delegates the construction of $PROG$ to another programmer, the left-hand sides and the type of $PROG$ being the interface between the two).

5.2.6. Within $PROG$ one may define other arithmetic functions, e.g.

$$\mathbf{df}\ eq0 : nat \to bool \ = \ nrec\,(bool,\ (\mathbf{fn}\, x : bool \bullet false),\ true) \bullet \ldots$$

assuming that the global identifiers *bool*, *false* and *true* have been defined properly in that context. Similarly one can adapt the expression $PRED$ to the G-typing, and use it to define $pred : nat \to nat$ within $PROG$.

5.2.7. An alternative type and definition for *nrec* is:

$$\begin{aligned}
&nrec' : (nat \to NAT) \\
&\qquad = \ \mathbf{fn}\, n : NAT \bullet \ (\mathbf{fn}\, z : \mathbf{tp},\ f : (z \to z),\ a : z \bullet n(z, f, a))
\end{aligned}$$

and the right-hand side may even be replaced by $\mathbf{fn}\, n : NAT \bullet n$ and $GID(NAT)$. But notice that with the definition

$$nrec'' : (nat \to nat) \ = \ \ldots\ldots \text{as for } nrec' \ldots\ldots$$

we cannot use $nrec''$ differently from the identity function on *nat*-expressions.

5.2.8. Let us abbreviate the sequence of the left-hand sides by SIG_{nat} and the sequence of right-hand sides by $IMPL_{NAT}$. (‘*SIG*’ is mnemonic for signature and ‘*IMPL*’ for implementation.) Then we may also write

$$(\mathbf{fn}\, SIG_{nat} \bullet PROG)(IMPL_{NAT})$$

and this is G-typed and equivalent to the previous program (w.r.t. both typing and evaluation). The expression shows more clearly that the implementation may be changed independently of the signature.

5.3. Parameterized Abstract Types: List of Elements. We shall construct something like “$list(elt)$” where *elt* is a parameter, in such a way that the construct “$list(elt)$” can be used with different choices for *elt*. The problem here is that “$list(elt)$” can not be a G-type, because we would then have $(\ldots \to \mathbf{tp})$ as type for *list* and such G-types do not exist. Nevertheless a satisfactory solution is possible and is easily generalized to, say, “$array(elt, n)$” for arrays of run-time determined fixed length n.

As for numbers it suffices to have nil (the empty list), cons (for constructing an element and a list into a new list) and lrec (for generic primitive recursion over lists) as the primitive operations and constant of lists. Head, tail, eqnil, append, map and so on can be defined in terms of them. (There is however no objection at all to enlarge the set of primitives.) We set

$$\begin{aligned} SIG \quad \equiv \quad & list : \mathbf{tp}, \\ & nil : list, \\ & cons : (elt, list \to list), \\ & lrec : (z : \mathbf{tp}, (elt, z \to z),\ z \ \to \ (list \to z)). \end{aligned}$$

So SIG gives the signature of the abstract data type of lists. It is not an expression, but a series of left-hand sides of definitions or formal parameters. Notice also that *elt* occurs globally in *SIG*; it will be used as the type of the list elements. For the time being we assume that we have an implementation for the signature, i.e. a series of expressions that we call *IMPL*:

$$IMPL \quad \equiv \quad LIST,\ NIL,\ CONS,\ LREC$$

where, again, identifier *elt* occurs globally in $LIST, \ldots, LREC$. We postpone the construction of *IMPL* and first focus on instantiating *IMPL* by different choices for *elt*.

Let *PROG* be a user program that computes with lists of numbers: within *PROG*, *list* is assumed to be a type, *cons* to be of type $(nat, list \to list)$ (rather than $(elt, list \to list)$), and similarly for *nil* and *lrec*. In short, *PROG* is G-typed under the typing assumptions $[nat/elt]SIG$. We may then form

$$(\mathbf{fn}\, [nat/elt]SIG \bullet PROG)([nat/elt]IMPL)$$

to obtain a G-typed expression with the desired behaviour. Now suppose that *PROG* uses both lists of *nat*s and lists of *bool*s. Let $[nat/elt]SIG'$ and $[bool/elt]SIG''$ be the assumptions under which *PROG* is G-typed. (By a single/double prime on *SIG* we mean that each of $list, \ldots, lrec$ gets a single/double prime.) As before we may now form

(1) $(\mathbf{fn}\ [nat/elt]SIG',\ [bool/elt]SIG'' \ \bullet\ PROG)\,([nat/elt]IMPL,\ [bool/elt]IMPL)$

to obtain a G-typed expression with the desired behaviour. However, we have duplicated *IMPL* and performed the substitutions *nat/elt* and *bool/elt* in *IMPL* manually. This is quite unsatisfactory, and can not claimed to be (a good model of) a practical programming language concept. Fortunately, there is better way by using parameterization. In the following expression 'AT' is mnemonic for 'abstract type', 'gen' for 'generic' or 'generate', and T is the type of *PROG*.

(2) $\mathbf{df}\ \mathit{genListAT} : \ (\mathit{elt} : \mathbf{tp},\ (\mathit{SIG}{\rightarrow}T) \rightarrow T)$
$\qquad = (\mathbf{fn}\ \mathit{elt} : \mathbf{tp},\ p : (\mathit{SIG}{\rightarrow}T) \bullet p(\mathit{IMPL}))$
$\bullet\ \mathit{genListAT}\,(\mathit{nat},\ (\mathbf{fn}\ [\mathit{nat}/\mathit{elt}]\mathit{SIG} \bullet \mathit{PROG}))$
respectively
$\mathit{genListAT}\,(\mathit{nat},\ (\mathbf{fn}\ [\mathit{nat}/\mathit{elt}]\mathit{SIG}'$
$\qquad \bullet\ \mathit{genListAT}\,(\mathit{bool},\ (\mathbf{fn}\ [\mathit{bool}, \mathit{elt}]\mathit{SIG}'' \bullet \mathit{PROG}))))$

So inside *genListAT* the user program receives the implementation, and thanks to the argument for parameter *elt* the implementation is suitably instantiated. Notice also that the nested call of *genListAT* is not at all recursive. There is yet one adaptation necessary; it concerns *PROG*'s result type T. As it stands, T is fixed within *genListAT* but naturally we want T to vary with the argument for p. Hence T should be made a parameter and we get:

(2′) $\mathbf{df}\ \mathit{genListAT} : \ (\mathit{elt} : \mathbf{tp},\ t : \mathbf{tp},\ (\mathit{SIG}{\rightarrow}t) \rightarrow t)$
$\qquad = (\mathbf{fn}\ \mathit{elt} : \mathbf{tp},\ t : \mathbf{tp},\ p : (\mathit{SIG}{\rightarrow}t) \bullet p(\mathit{IMPL}))$
$\bullet\ \mathit{genListAT}\,(\mathit{nat},\ T,\ (\mathbf{fn}\ [\mathit{nat}/\mathit{elt}]\mathit{SIG} \bullet \mathit{PROG}))$
and so on...

Actually the type-checker may deduce T from *PROG* and the programmer need not write it explicitly.

Notes

5.3.1. One might now go on and design new expression forms for the definition and use of abstract types. This has been done indeed, and gives rise to the introduction of a ∀- and a ∃-type and a special syntax for program scheme (2′). Essentially, $\forall\, \mathit{elt} \bullet t$ is a generic type and abbreviates $(\mathit{elt} : \mathbf{tp} \rightarrow t)$, whereas $\exists\, \mathit{elt} \bullet t$ is a generic signature and abbreviates $\mathit{elt} : \mathbf{tp},\ t$ (where t may be a cartesian product $t_1, t_2, \ldots, t_n$). See Cardelli & Wegner [4] and Mitchell & Plotkin [21]. A formal Representation Independence has been proved for the typing system of [21], see [22].

5.3.2. Consider once more expressions (1) and (2′) and in particular type T of *PROG*. In (1), T may be expressed in terms of list' and list'' and there is no problem whatsoever with the G-typability of the entire expression (1). E.g., if $T \equiv \mathit{list}'$ then (1) has type $[\mathit{nat}/\mathit{elt}]\mathit{LIST}$; cf. also Note 5.2.5. Within (2′) however it seems impossible to arrange that $T \equiv \mathit{list}'$. The reason is that T falls outside the scope of $[\mathit{nat}/\mathit{elt}]\mathit{SIG}$, i.e. T is not in the scope of the locally defined *list*. This forms *our* explanation of the requirement AB.3 in [21], viz. that a program may not deliver a value of a locally defined abstract type.

5.3.3. Constructions like "list of list of elements" are possible too. For example, replace in (2) *bool* by list' (i.e. list of *nat*s).

5.3.4. Within *PROG* other list manipulating functions can be defined, thereby using the entries of $[\mathit{nat}/\mathit{elt}]\mathit{SIG}$. For example

$$\begin{aligned}\mathbf{df}\ \ hd\ &:\ (\mathit{list} \to \mathit{nat})\\ &=\ \mathit{lrec}\,(\mathit{nat},\ (\mathbf{fn}\, x : \mathit{nat},\ y : \mathit{nat} \bullet x),\ \mathit{DONTCARE})\end{aligned}$$

defines the head-function for lists of numbers (with *DONTCARE* determining the outcome for "the head of the empty list *nil*").

It remains to construct some *IMPL*, i.e. some *LIST*, *NIL*, *CONS*, *LREC*. The construction below is quite analogous to the definitions of *nat*, *zero*, *succ* and *nrec* given in Section 5.2, and follows the suggestion of Section 3.6. Here are the type and expressions:

$$\begin{array}{lcl} \mathit{LIST} & \equiv & (z : \mathbf{tp},\ (\mathit{elt}, z \to z),\ z\ \to z) \\ \mathit{NIL} & \equiv & \mathbf{fn}\, z : \mathbf{tp},\ f : (\mathit{elt}, z \to z),\ a : z \bullet a \\ \mathit{CONS} & \equiv & \mathbf{fn}\, x : \mathit{elt},\ l : \mathit{LIST} \\ & & \bullet\ \ (\mathbf{fn}\, z' : \mathbf{tp},\ f : (\mathit{elt}, z' \to z'),\ a : z'\ \bullet f(x,\ l(z', f, a))\,) \\ \mathit{LREC} & \equiv & \mathbf{fn}\, z : \mathbf{tp},\ f : (\mathit{elt}, z \to z),\ a : z\ \bullet\ (\mathbf{fn}\, l : \mathit{LIST} \bullet l(z, f, a)). \end{array}$$

5.4. Modules. For large scale programs modularity is of utmost importance. The wide variety in modern programming languages is substantially due to the constructs for modularity: **package**s in Ada, **module**s in Modula 2, **cluster**s in CLU, **program**s in Modular Pascal and so on. We shall express a very general module concept in the Lambda Calculus and design a new syntactic form for this particular expression scheme. For simplicity we do not consider typing.

In order to demonstrate the generality (not to *express* the concept) we assume in this subsection that the Lambda Calculus has been extended with assignment, assignable variables, sequencing and, if you wish, exception handling. (These extensions surely invalidate so much of the properties of the Lambda Calculus that no one would ever call it "Lambda Calculus" any more.)

The example problem that we tackle is a classical one: it is requested to write a "module" for a random number generator that allows the user to specify the "seed" (which determines the pseudo-random sequence completely) and that "exports" a parameterless function for "drawing" a next random number from the sequence. It should also be possible that several instantiations of the module be active simultaneously for several independent pseudo-random sequences.

It is known that $a_0, a_1, a_2, \ldots$ is a pseudo-random sequence if, for some suitable constants m and d, we have $a_i\ =\ a_{i-1} {\times} m\ \mathbf{mod}\ d$ for all $i > 0$; a_0 is the seed. Therefore we wish to construct the solution from the following three ingredients:

$\mathbf{var}\ a : \mathit{nat} \bullet\ \ldots\ldots$ local store for the a_i
$a := \mathit{seed}$ initialization
$\mathit{DRAW} \equiv (\mathbf{fn}\,() \bullet\ a :=\ a {\times} m\ \mathbf{mod}\ d;\ \text{result is}\ \ a)$
function that yields the next random number

There are two main problems: *to control the visibility* so that the scope of $\mathbf{var}\, a$ does not extend over the user's program, and *to control the life-time* of $\mathbf{var}\, a$ so that storage is allocated for a precisely during the evaluation of the user's program *PROG*.

Quite surprisingly our successful attempts to express parameterized abstract types in the G-typed Lambda Calculus provide already the solution. In expressions (2) and (2′) above, *IMPL* is invisible in *PROG even if all typing were omitted*! Moreover, had there been Pascal-like variables in the body of *genListAT*, these would exist as long as the evaluation of p (and therefore of *PROG*) would last. Thus we find:

```
df rng = (fn seed, p • var a : nat • a := seed; p (DRAW))
 • ......
   rng (041130, (fn draw • PROG))
   respectively
   rng (041130, (fn draw • rng (161087, (fn draw′ • PROG′))))
   ......
```

It seems worthwhile to design a special syntactic form for the above scheme: module expressions and module invocations. First we show their use and then we define them formally. Here is the above program written with the module constructs.

```
df rng = (fn seed • module
                       var a : nat • a := seed; export(DRAW)
                     endmodule)
 • ......
   (invoke draw = rng (041130) • PROG)
   respectively
   (invoke draw = rng (041130) •  (invoke draw′ = rng (161087) • PROG′))
   ......
```

The module consists of an expression in which one subexpression is tagged with **export**. An **invoke**-expression is syntactically similar to a **df**-expression. The evaluation of **invoke** $x = em \bullet eb$ consists of evaluating the module expression em after replacing the part **export**(ea) in it by **df** $x = ea \bullet eb$. Thus it may be better to say that eb is imported into em rather than that ea is exported to eb. Formally, we consider the left-hand sides, below, as abbreviations for the right-hand sides:

module .....**export**(ea).... **endmodule**	**fn** $p \bullet$ ..... $p(ea)$....
invoke $x = em \bullet eb$	em (**fn** $x \bullet eb$)

Hence, the derived evaluation rule reads:

replace	**invoke** $x =$ (**module**.....**export**(ea).... **endmodule**) $\bullet\ eb$
by	.....(**df** $x = ea \bullet eb$)....

Notes

5.4.1. One may, of course, replace the fragment $a := seed;\ p(DRAW)$ by **df** $drw = DRAW \bullet a := seed;\ p(drw)$. It thus turns out that the (first) solution can be transliterated to Pascal, so that in principle no extra module construct is needed in Pascal.

5.4.2. A formal proof that "storage for a is allocated precisely during the evaluation of *PROG*" can not be given before assignment and variables have been added formally to the Lambda Calculus.

5.4.3. Neither Pascal-like dynamically allocated variables, nor Algol 68 **heap** variables, facilitate a solution to the problem of controlling the life-time of a satisfactorily. Algol 60 has the concept of **own** variable for this purpose. But, whereas the interference of recursion and **own** variables gives problems in Algol 60, there are no such problems here (because both recursion and the above solution are expressed entirely within the Lambda Calculus).

5.4.4. Generalization to multiple export and invocation is straightforward.

5.4.5. Not only initialization "before the export" is possible, but also finalization "after the export", and even exception handling "around the export". For the latter, imagine an "exception" defined locally within the module, possibly "raised" from within $DRAW$ when used in $PROG$, and "handled" at/around the export expression within the module. Other arrangements are possible too, e.g. exporting a locally defined "exception" jointly with $DRAW$ so that it may be handled from within $PROG$ as well. Also, by a slight adaptation of the *rng* definition, we get a module that yields *initialized variables*: export a itself rather than $DRAW$. For details see Fokkinga [7].

6. Beyond Generic Typing

As shown informally in Section 5, generic typing is quite expressive; a precise characterization of the G-typable arithmetic functions is given by Fortune et al. [9]. Nevertheless there are reasons for further generalization:

- There still exist expressions that are semantically meaningful but not G-typable; e.g. $TW(TW)(K)$ where $TW \equiv \mathbf{fn}\, f \bullet (\mathbf{fn}\, x \bullet f(f(x)))$ and $K \equiv \mathbf{fn}\, x \bullet (\mathbf{fn}\, y \bullet x)$, [10].
- Functions like $\mathbf{fn}\, n, x_1, \ldots, x_n \bullet x_1 + \ldots + x_n$ for which the first argument determines the number of following arguments, are not G-typable.
- Referring to *SIG* and *IMPL* of Section 5, it seems natural to make the implementation *IMPL* into one tuple expression $\langle IMPL \rangle$, from which the individual components can be retrieved by selections .1, .2, ...; the type of $\langle IMPL \rangle$ would then be a tuple type $\langle SIG \rangle$. (Notice the *dependency* between the first and following components within $\langle SIG \rangle$.)
- One might wish to extend the type formation rules in such a way that arbitrary properties can be expressed in types, and typability means total correctness with respect to the properties expressed in the types.

Much work is, and has been, done towards the fulfillment of the last point above: the AUTOMATH project [2], Martin-Löf's Intuitionistic Theory of Types [16], and recently the Theory of Constructions [5]. Space limitations do not permit us to discuss these very promising approaches. Instead, we briefly present our own devised typing, called SVP-typing [8]. Due to its far going generalization, it is quite simple to define but, as a price to be paid, has some weak points that have been avoided consciously in AUTOMATH, the Intuitionistic Theory of Types and the Theory of Constructions:

- there is no distinction any more between types and normal value expressions;
- the evaluation of typed expressions may not terminate.

Consequently, compile-time type-checking may sometimes not terminate. This is really a pity, but hopefully not disastrous:

- we expect that type errors will be detected far more often than that the type-checker does not terminate;
- nontermination of the type-checker can be treated in the same way as nontermination of programs nowadays: an unexpectedly long type-checking time should make someone suspicious and suggests to prove termination or change the program (or typing) otherwise.

It is left open for future research whether this is a sensible approach.

6.1. SVP-Typing. The SVP-typing is a generalization of the G-typing that was already anticipated when we designed the syntax for generic constructs. Basically, types are now merely expressions that have type $\mathbf{tp}$. In particular they may be the result of functions and components of tuples, and $\mathbf{tp}$ itself is a type (so that $\mathbf{tp}$ has type $\mathbf{tp}$).

SVP′-typed expressions. The following grammar generates the SVP′-typable expressions. In each rule we distinguish constituent parts by suffixes f, a, b, x and r, and we stipulate that equally named constituents are equal. Moreover, for readability we write 't' for '$e^{\mathbf{tp}}$', 'tx' for '$ex^{\mathbf{tp}}$', 'tr' for '$er^{\mathbf{tp}}$', and so on.

$$\begin{array}{lll}
t & ::= \mathbf{tp} & \text{the type of types} \\
e^{tx} & ::= x^{tx} & \\
t & ::= (x^{tx} : tx \to tr) & \text{the type of functions} \\
e^{x:tx\to tr} & ::= (\mathbf{fn}\, x^{tx} : tx \bullet\ eb^{tr}) & \\
e^{[ea/x]tr} & ::= ef^{x:tx\to tr}(ea^{tx}) & \\
t & ::= \langle x^{t_1} : t_1,\, t_2\rangle & \text{the type of tuples} \\
e^{\langle x:t_1,t_2\rangle} & ::= \langle e_1^{t_1},\, e_2^{[e_1/x]t_2}\rangle & \\
e^{t_1} & ::= ep^{\langle x:t_1,\, t_2\rangle}.1 & \\
e^{[ep.1/x]t_2} & ::= ep^{\langle x:t_1,\, t_2\rangle}.2 &
\end{array}$$

An expression of type $\mathbf{tp}$ is called a *type*; we let t, ta,... denote arbitrary types. Within $(x : t' \to t'')$, $(\mathbf{fn}\, x : t' \bullet eb^{t''})$ and $\langle x : t', t''\rangle$ identifier x is a local name whose scope extends over t'' and eb, but not over t'. If x does not occur in t'', we simply write $(t' \to t'')$, respectively $\langle t', t''\rangle$.

Notes
6.1.1. The evaluation rules and the generalization to multiple parameters, definitions and tuples are obvious and tacitly used in the sequel.

6.1.2. Due to the last rule the type of an expression is not uniquely determined.

6.1.3. Clearly, the SVP′-typing subsumes the G-typing. Thanks to the concrete notation that we have designed both a G-type and a G-typed expression are SVP′-typed expressions.

6.1.4. Functions that yield types, and tuples that contain types, are now possible. For instance, with *SIG* and *IMPL* of the previous section, we may now write:

$$\mathbf{df}\ genListAT' : (elt : \mathbf{tp}\ \to \langle SIG\rangle)\ =\ (\mathbf{fn}\ elt : \mathbf{tp} \bullet\ \langle IMPL\rangle) \bullet\ \ldots\ldots$$

On the dots $genListAT'(nat)$ has type $[nat/elt]\langle SIG\rangle$ so that, according to the typing rules for tuple selection:

$genListAT'(nat).1$	$(\equiv NLIST)$ has type $\mathbf{tp}$ and is the representation type for lists,
$genListAT'(nat).2$	has type $NLIST$ and is the nil for lists of numbers,
$genListAT'(nat).3$	has type $(nat, NLIST\ \to NLIST)$ and is the cons for lists of numbers, and
$genListAT'(nat).4$	has type $(z : \mathbf{tp},\ (z \to z),\ z\ \to (NLIST \to z))$ and is the lrec for lists of numbers.

6.1.5. The dependency has been generalized too. Not only type parameters and components may be referred to in later parameters, result and components, but also normal value parameters and components; e.g. $sort : (n : nat,\ elt : \mathbf{tp},\ a : array(elt, n)\ \to\ array(elt, n))$. This kind of dependency has been strived for in the design of PEBBLE, a typing system for large scale modularity [3].

6.1.6. Given *array* of type $(elt : \mathbf{tp},\ n : nat\ \to \mathbf{tp})$, as above, we find that

$$ef^{(a:array(bool,7) \to \ldots)}(ea^{array(bool,3+4)})$$

is not SVP′-typed: the rule for function call requires that the parameter type and the argument type be syntactically equal. This leads to the extension below.

SVP″-typed expressions. The grammar consists of all rules for the SVP′-typed expressions, and in addition:

$$e^{t'}\ \ ::=\ \ e^{t''} \qquad \text{whenever } t' \text{ and } t'' \text{ are semantically equivalent.}$$

(There are several ways to define semantic equivalence; one way is to say that expressions are semantically equivalent if they can be evaluated to a common intermediate result.)

Notes
6.1.7. It is the very combination of this rule with $\mathbf{tp}^{\mathbf{tp}}$ that seems to allow for SVP″-typed expressions with nonterminating evaluations; cf. Meyer & Reinhold [19].

6.1.8. We conjecture that it is impossible to express the tuple constructs in the others, i.e. to replace the tuple constructs by SVP″-typed equivalent functions.

6.1.9. Now that evaluation on type positions (superscripts) is allowed, we can reformulate the grammar rules for function constructs:

$$t \quad ::= \quad (\mathbf{fn}\, x : \mathbf{tp} \bullet tr) \tag{1}$$
$$e^{\mathbf{fn}\, x:\mathbf{tp}\bullet tr} \quad ::= \quad (\mathbf{fn}\, x^{tx} : tx \bullet eb^{tr}) \tag{2}$$
$$e^{tf(ea)} \quad ::= \quad ef^{tf^{(\mathbf{fn}\, x:\mathbf{tp}\bullet\mathbf{tp})}}(ea^{tx}) \tag{3}$$

That is, $\mathbf{fn}\, x : tx \bullet tr$ plays the role of the type $(x : tx \to tr)$; it contains the same information and, indeed, $(\mathbf{fn}\, x : tx \bullet tr)(ea)$ is semantically equivalent to $[ea/x]tr$.

6.1.10. According to rule (1) above $\mathbf{fn}\, x : t' \bullet t''$ has type $\mathbf{tp}$, and according to rule (2) [taking tx, eb, tr to be $t, t', \mathbf{tp}$] it has type $\mathbf{fn}\, x : tx \bullet \mathbf{tp}$ too. This suggests to replace rule (1) by

$$
\begin{array}{lll}
e^{t'} & ::= & e^{t''} \quad \text{whenever } t'' \leq t' \\
\leq & \text{is} & \text{the reflexive transitive closure generated by } (\mathbf{fn}\, x : tx \bullet eb^{tr}) \leq (\mathbf{fn}\, x : tx \bullet tr)
\end{array}
$$

This approach has been studied in the context of AUTOMATH and is incorporated in some version of the Theory of Constructions.

7. Concluding Remarks

Much of the programming language concepts that we have discussed, deal with the —intuitive— notion of "abstraction", which is to neglect, consciously, some aspects of the subject under consideration. It turns out that the **fn**-construct facilitates this abstraction. Since the syntactic manipulation of forming $\mathbf{fn}\, x \bullet e$ out of e and x, is called Lambda-abstraction, we conclude from our exposition that

Lambda-Abstraction is the key to Intuitive Abstraction.

Many programming language concepts have not been discussed here, notably assignment and assignable variables, and exception handling. These two concepts in particular require a drastical extension/change of the Lambda Calculus. In [7] we have done so, and it turns out that the formalism does not change much; the properties do. There we also show how the conventional stack-based implementations may be derived in a systematic way from the the "replacement" semantics of the Lambda Calculus. Thus the applicability of the Lambda Calculus approach to programming language concepts is wider than sketched in this paper.

Finally we remark that one should not confuse "programming language concepts" with "programming concepts".

Acknowledgement. I have had much profit from our recent study group on Lambda Calculus and, earlier, from discussions with Mirjam Gerritsen and Gerrit van der Hoeven. It has been a great stimulus for me to know that this paper would be dedicated to Leo Verbeek.

References

1. H.P. Barendregt: *The Lambda Calculus — Its Syntax and Semantics*, Studies in Logic 103, North-Holland, Amsterdam, 1981. (2nd edition 1984)
2. N.G. de Bruijn: A survey of the Automath project, *in*: J.P. Seldin & J.R. Hindley (Eds.): *To H.B. Curry — Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London, 1980.
3. R. Burstall & B. Lampson: A kernel language for abstract data types and modules, *in*: G. Kahn, D.B. MacQueen & G. Plotkin (Eds.): *Semantics of Data Types*, Lect. Notes Comp. Sci. **173** (1984) 1-50, Springer-Verlag, Berlin - Heidelberg - New York.
4. L. Cardelli & P. Wegner: On understanding types, data abstraction, and polymorphism, *Comput. Surveys* **17** (1985) 471-522.
5. Th. Coquand & G. Huet: Constructions — a higher order proof system for mechanizing mathematics, *EUROCAL 85*, Lect. Notes Comp. Sci. **203** (1985) 151-184, Springer-Verlag, Berlin - Heidelberg - New York.

6. Th. Coquand & G. Huet: A selected bibliography on constructive mathematics, intuitionistic type theory and higher order deduction, *J. Symbolic Comput.* **1** (1985) 323-328.

7. M.M. Fokkinga: *Structuur Van Programmeertalen*, University of Twente, Enschede, Netherlands, 1983. (Lecture Notes in Dutch, "Structure of Programming Languages".)

8. M.M. Fokkinga: *Over het nut en de mogelijkheden van typering*, University of Twente, Enschede, Netherlands, 1983. (Lecture Notes, in Dutch, "On the use and the possibilities of typing".)

9. S. Fortune, D. Leivant & M. O'Donnell: The expressiveness of simple and second order type structures, *J. Assoc. Comput. Mach.* **30** (1983) 151-185.

10. M. Gerritsen & G.F. van der Hoeven: Private communication, 1987.

11. J.R. Hindley & J.P. Seldin: *Introduction to Combinators and Lambda Calculus*, London Mathematical Society Student Texts **1**, Cambridge University Press, Cambridge (U.K.), 1986.

12. P.J. Landin: The mechanical evaluation of expressions, *Computer J.* **6** (1964) 308-320.

13. P.J. Landin: A correspondence between Algol 60 and Church's Lambda notation, *Comm. Assoc. Comput. Mach.* **8** (1965) 89-101, 158-165.

14. P.J. Landin: The next 700 programming languages, *Comm. Assoc. Comput. Mach.* **9** (1966) 157-166.

15. H. Langmaack: On procedures as open subroutines I, *Acta Inform.* **2** (1973) 311-333.

16. P. Martin-Löf: An intuitionistic theory of types: predicative part, *in*: *Logic Colloquium 1973*, pp. 73-118, North-Holland, Amsterdam, 1975.

17. L.G.L.T. Meertens: Procedurele datastructuren, *in*: *Colloquium Datastructuren*, Mathematisch Centrum (Currently CWI), Amsterdam, 1978.

18. L.G.L.T. Meertens & S. Pemberton: Description of B, *ACM SIGPLAN Notices* **20** (1985) 58-76.

19. A.R. Meyer & M.B. Reinhold: 'Type' is not a type — Preliminary Report, *in*: *ACM Conf. Record of the 13th Annual Symposium on Principles of Programming Languages* **13** (1986) 187-295.

20. R. Milner: A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978) 348-375.

21. J.C. Mitchell & G.D. Plotkin: Abstract types have existential type, *in*: *ACM Conf. Record of the 12th Annual Symposium on Principles of Programming Languages* **12** (1985) 37-51.

22. J.C. Mitchell: Representation independence and data abstraction (preliminary version), *in*: *ACM Conf. Record of the 13th Annual Symposium on Principles of Programming Languages* **13** (1986) 263-276.

23. P. Naur (Ed.): Revised report on the algorithmic language ALGOL 60, *Comm. Assoc. Comput. Mach.* **6** (1963) 1-17.

24. J.C. Reynolds: Towards a theory of type structure, *in*: B. Robinet (Ed.): *Programming Symposium*, Lect. Notes Comp. Sci. **19** (1974) 408-425, Springer-Verlag, Berlin - Heidelberg - New York.

25. J.C. Reynolds: User-defined data types and procedural data structures as complementary approaches to data abstraction, *in*: S.A. Schuman (Ed.): *New Directions in Algorithmic Languages 1975*, pp. 154-165, IRIA, France, 1976.

26. J.C. Reynolds: The essence of ALGOL, *in*: J.W. de Bakker & J.C. van Vliet (Eds.): *Algorithmic Languages*, pp. 354-372, North-Holland, Amsterdam, 1981.

27. J.C. Reynolds: Polymorphism is not set-theoretic, *in*: G. Kahn, D.B. MacQueen, G. Plotkin (Eds.): *Semantics of Data Types*, Lect. Notes Comp. Sci. **173** (1984) 145-156, Springer-Verlag, Berlin - Heidelberg - New York.

28. J.C. Reynolds: Three approaches to type structure, *in*: H. Ehrig et al. (Eds.): *Mathematical Foundations of Software Development*, Lect. Notes Comp. Sci. **185** (1985) 97-138, Springer-Verlag, Berlin - Heidelberg - New York.

29. A. Rezus: *A Bibliography of Lambda Calculi, Combinatory Logics and related topics*, Mathematisch Centrum, Amsterdam, 1982.

30. R.D. Tennent: *Principles of Programming Languages*, Prentice Hall, 1981.

31. D. Turner: Miranda — a non-strict functional language with polymorphic types, *in*: *Proc. Int. Conf. on Functional Programming Languages and Computer Architecture*, Lect. Notes Comp. Sci. **201**, (1985) 1-16, Springer-Verlag, Berlin - Heidelberg - New York.

32. D. Turner: An overview of Miranda, *ACM SIGPLAN Notices* **21** (1986) 158-166.