Endomorphic Typing

Michel Sintzoff

Université Catholique de Louvain, Unité d'Informatique,
pl. Ste-Barbe 2, B-1348 Louvain-la-Neuve, Belgium

Abstract

This paper surveys a number of requirements for the adequate formulation of deductions such as program derivations. The kernel of a possible framework for such formulations is proposed. It is described in terms of a naive semantics and of corresponding algebras; deductions are composed from basic deduction-rules, and typing is defined as an endomorphism on deductions. Approaches based on typed λ-calculi, including the Deva language, are compared with the proposed one, and a few general issues are discussed.

1 Expression of Program Derivations

First, we sum up requirements for the expression of program derivations. These requirements stem from experiments in the formalization of program deductions and of programming methods. Accordingly, the initial driving force has been programming rather than logic or algebra; this may explain differences in basic options. Yet, logic and algebra played a growing part in the subsequent technical elaboration, and the resulting approach appears to be also relevant for proofs.

1.1 Deductions of Programs

Given the Curry-Howard isomorphism between types and formulae [5] (§9E), it is mathematically legitimate to derive programs by extracting recursive functions from constructive proofs. This characterizes one specific method of program design. Our aim is to express also other mathematical methods of program construction, such as refinements and transformations.

As a consequence, we relate programs to deductions as follows. Problem specifications and the corresponding programs are all viewed as formulae at the same level of discourse. The derivations yielding programs from specifications are then considered to be deductions of formulae from formulae.

1.2 Specifications of Deductions

Program deductions should not be recorded just in terms of constructive steps, e.g. actual rewritings, substitutions, or applications of laws: without knowing the desired effect of such concrete steps, it is difficult to understand the logic of deductions. Thus, operational steps in program deductions should themselves be specified, e.g. in terms of antecedent prerequisites and consequent properties [21]. This amounts to using at the meta-level the technique of specifying program components by pre- and post-conditions: deductions yielding programs are meta-programs, and the specifications of these deductions are meta-specifications. If a deduction step is specified by antecedent and consequent formulae, this step actually realizes an inference rule consisting of that antecedent and that consequent.

It is well recognized that programs should be given together with their specifications, and should be correct w.r.t. these. Similarly, program deductions and programming methods should be correct w.r.t. their own specifications. A programming method abstracts a family of similar deductions; it is specified by the corresponding families of source specifications and of target programs.

As a consequence, two initial requirements emerge:

- elementary and composed deductions should be correct w.r.t. pairs of formulae;
- deductions and formulae should be expressible at various levels, e.g. for programs, for rules of program deduction, and for deductions of deduction rules.

1.3 Scaling-up of Deductions

Applications demonstrate the importance of structuring means in expressing program deductions [22]. The need for constructs which are already well known at program level has been naively re-discovered at the level of program deductions. The following characteristics appear essential for effective scaling-up and intellectual mastery:

- compositionality: deductions should be composed serially, collaterally, and iteratively, viz by functional composition, by product, and by recursion;
- definitionality: auxiliary deductions should be definable in abbreviations;
- genericity: deductions should be parametrizable w.r.t. formulae and their properties;
- inheritance: contexts in which deductions are formulated should themselves be structured and re-usable, also using abstractions, compositions, definitions, and genericity;
- economy: identical structures, e.g. composition, at different levels should use identical means of expression;
- consistency: the formalization framework may not cause new logical errors.

1.4 Publishable Deductions

A careful distinction must be made between the ideal expression of the ideal deductions producing a given program, and the faithful expression of the discoveries, searches, attempts, heuristics, hints, and failures underlying the elaboration of those ideal deductions. The present requirements concern the former and not the latter: these issues must be separated, lest they become unmanageable. In other words, our current aim is merely to find formal means of expressing completed deductions in reference publications so that, by their adequacy and attractiveness, they would favourably compare with the current semi-formal techniques. This aim may be far-off, but is not far-fetched in principle: after all, the old semi-formal style of publishing numerical algorithms has been successfully replaced by the structured use of clear algorithmic notations.

As a consequence, our aim is not primarily to assist thinkers in discovering deductions, but more modestly to provide them with a formal and attractive framework in which to communicate these deductions in a clear setting. By "attractive", we mean "better than with usual means". The state of the art in formal notations for deductions is unsatisfactory in this respect: the formalization of semi-formal deductions typically multiplies their length by a significant factor, and often obscures the reasoning instead of clarifying it. The elegance of exemplary semi-formal deductions [25] should be preserved, and the expansion factor should rather be a reduction factor. Algebra itself has evolved this way.

By semi-formal deductions, we refer to the current representations of deductions by sequences, trees or graphs; in the latter case, the vertices are intermediate formal expressions, and the edges are informal indications. These informal edges should be formalized in such a way that the formulae on vertices could be left out at will, and that the deductions would become nice as well as precise.

Besides clarity and structure in the expression of program deductions, additional useful effects can be obtained. For instance, formal deductions can be verified and processed by algorithms; we all know the pitfalls of hasty, semi-formal deductions. Thus, automated assistance in maintaining correctness and in performing standard sub-deductions can certainly be provided. But the wish to assist should not hamper the fundamental aims of elegance and structure in the final expressions of deductions. Let us avoid thinker-hostile systems.

2 Naive View of a Framework of Expression

The requirements above, even excluding automated assistance, are challenging; we try to satisfy these by successive approximations, being unable to do it at once. We thus present yet another approximation in this direction, and focus on its semantics so as to discuss fundamental issues. Semantics means here understanding, not primarily mathematical theories or operational models. The concepts presented below have been identified and selected on the basis of experiments in the expression of program deductions using various methods.

A deduction step realizes a correspondence between formulae. Such formulae may express problem-specifications and programs. So, a central concept is that of an effective derivation step c implementing a **rule** from formula a to formula b, in short $c : a \mapsto b$. The Curry-Howard isomorphism between types and formulae

$$\frac{types}{functions} = \frac{formulae}{deductions}$$

is thus instantiated to the following relationship:

$$\frac{types}{functions} = \frac{formulae\ for\ specifications\ and\ programs}{deductions\ of\ programs\ from\ specifications}\ .$$

The type Tc of a deduction c has the structure of an inference rule from a problem-specification a to a program b, viz $Tc = (a \mapsto b)$. This constitutes the essence of **correctness** for a deduction c.

In general, $Tu = v$ means that u realizes, implements, or substantiates v; in other words, v is the class or the type of u. If a and b respectively belong to the classes t and s, then the product $a \times b$ belongs to the class pairing t and s, viz the product $t \times s$. Accordingly, a rule or correspondence from a to b, which is also a pair, belongs to the class which is a correspondence from t to s. The type of a construction has thus the **same structure** as this construction.

The types of derivations can be viewed as abstract derivations, and may themselves be given more abstract types. The concept of abstractness is *relative*, since any classifying entity can in turn be classified. For instance,

$$T(8192) = Natural, \quad T(Natural) = Algebra, \quad T(Algebra) = Institution \ .$$

Deductions should be composed from basic deduction-rules by serial composition, collateral composition, case-analysis, and repetitions. Hence, other important concepts are those of *composition, product, sum,* and *iteration.* Functional composition appears more fundamental than function application to the extent that priority is given to the **composition** of deductions.

Deductions are thus structured very much like functions or programs. As function-expressions are formulae, deduction-expressions should also be taken as formulae, in order to express and to specify the deductions of deduction-rules ; this need has been repeatedly stressed in the context of program derivation [18, 23]. Hence, a useful concept is that of *higher-level formulae for* expressing *deductions* or properties of formulae. We adopt a deduction-level view from the outset, for the same reason we prefer function-level expressions for programs.

Environments define specific theories in which given deductions are carried out. Such contexts should be tackled as formal entities on a par with deductions; thus, *formulae for contexts* should be as well composed as formulae for deductions. Contexts and deductions should be definable *generically*, viz using *abstraction* and *instantiation. Inheritance* of available contexts and deductions is essential for an understandable composition of new ones; inheritance must be supported at least by the introduction and the use of *definitions* with *scope rules.*

Confusion between semantic levels must be prevented, e.g. by a *safe stratification* of formulae. In any case, **consistency** must be preserved.

The economy of fundamental structures is fostered by *unifying redundant variants* of identical syntactical operators, e.g. binding in functional expressions and binding in dependent products, and by *using dualities* whenever appropriate, e.g. between products and sums.

The above concepts are closer to the spirit of well-structured higher-level programming languages, than to the style of basic logic languages where structuring means usually are lacking or are introduced as extensions. There is a similar difference between an ML-like language and a basic typed-λ-calculus.

3 Algebraic Definition of the Proposed Framework

The view presented in the previous section can be formulated in terms of simple algebras. These will be presented gradually, by successive enrichments. In these algebras, typing is defined as an **endomorphism**: it is an homomorphism because types result from abstract deductions which have the same structure as the deductions to be typed; this homomorphism is internal, i.e. it is an endomorphism, because types are values too, albeit somewhat more abstract than other ones.

3.1 The Basic Algebras CT0: Composition of Rules and Typing

The first algebras we introduce are quite simple: they merely allow one to compose objects which in general have a rule-like structure. The CT0-algebras embody the main features of the proposed approach, namely composition of rules and endomorphic typing. The addition of other operations does not modify this kernel.

The presentation of the CT0-algebras consists of the following signature and axioms; the axioms for typing are given separately so as to clarify the definition.

Signature

Objects, i.e. CT0-elements, are defined as follows:
- constant objects are given;
- if a, b are objects, then $rule(a, b)$ and $comp(a, b)$ are objects, termed "composite";
- there are no other objects.

This amounts to the following signature, where the constants α_j $(j \in J)$ are nullary functions:

$$\begin{aligned} \alpha_j : \quad & \quad \longrightarrow object \\ rule,\ comp : object \times object & \longrightarrow object \end{aligned}$$

The following left-to-right notations are used:

$comp(a, b)$	$a; b$
$rule(a, b)$	$a \mapsto b$

A composite rule-object $a \mapsto b$ expresses a binary correspondence between the objects a and b. These correspondences are relational rather than functional, since $a \mapsto b$ as well as $a \mapsto c$ are allowed for any objects a, b, c. The recursive construction of objects by $\mapsto$ mirrors that of simple functional types.

Axioms

The operation *comp* is associative, and the operation *rule* is transitive:

$$a; (b; c) = (a; b); c \qquad (1)$$

$$(a \mapsto b); (b \mapsto c) = (a \mapsto c) \qquad (2)$$

Thus, when b and d are distinct objects, $(a \mapsto b); (d \mapsto c)$ is defined but is not reducible. The transitivity of $\mapsto$ mirrors the transitivity of implication and of rewriting.

The typing endomorphism

The typing endomorphism T maps objects to objects and it preserves the structure of CT0:

$$T : object \rightarrow object$$

$$T(a; b) = (Ta; Tb) \tag{3}$$
$$T(a \mapsto b) = (Ta \mapsto Tb) \tag{4}$$

In CT0, rule-objects $a \mapsto b$ are defined for any objects a, b. To characterize *realizable* rule-objects, we may use the typing endomorphism T: a rule-object $a \mapsto b$ is viewed as realizable, viz valid or inhabited, if it is is in the range of T, i.e. if there is some more concrete object c such that $Tc = (a \mapsto b)$. This c can be seen as a construction realizing the rule-type $a \mapsto b$.

The gist of the typing endomorphism is the correspondence between concrete derivations such as $(a; b)$ and their abstract counterparts such as $(u; v)$, which may well be $(p \mapsto q); (q \mapsto s) = (p \mapsto s)$.

Constraints on the typing endomorphism

The definition of typing as an endomorphism mainly says it preserves the structure of objects, i.e. T is stable through *rule* and *comp*. This may be seen as too permissive: for instance, it may be unwise to accept cycles through repeated typing, viz $T^3a = a$. On the other hand, it may be useful to have a root-object r which is its own type, viz $Tr = r$; such a root may serve as a terminal object w.r.t. typing.

From a mathematical standpoint, cycles may well exist in endomorphisms as in graphs. From a methodological standpoint, it may be wiser to consider only pure hierarchies in the abstraction levels of types. To this end, we may impose additional constraints on the typing endomorphism. For instance, in order to exclude cycles in typing, we can use typing to **stratify** objects as follows.

Among the constant objects, there is a **root**-object r such that $Tr = r$. This root-object ensures T is always defined. If we want to forbid typing cycles also on the root, we may simply introduce a denumerable set of roots r_i such that $Tr_i = r_{i+1}$ for $i \geq 0$.

Then, each object is given a **level** which measures its distance from below to the root-object by repeated typing. Levels stratify objects; at the top of the hierarchy, we have the objects of level zero, or *skeletons*, which are composed of the root-object only.

The level of any object a is the smallest integer n such that $T^n a$ is a skeleton. Thus,

$$level(a; b) = level(a \mapsto b) = \max(level(a), level(b)) \ .$$

The greater the level of an object, the more concrete or the more informative it is. The root-object gives almost no information. A stratifying endomorphism T decrements levels; this prevents semantic confusion between objects in different layers of abstraction.

For instance, if

$$Tc = a,\ \ Ta = r,\ \ Tr = r,\ \ Tb = (a \mapsto r)\ ,$$

then the level of $((a \mapsto c) \mapsto b)$ is 2 and its skeleton is $((r \mapsto r) \mapsto r)$:

$$\begin{array}{ll} & T^2\ ((a \mapsto c) \mapsto b) \\ = T & ((r \mapsto a) \mapsto (a \mapsto r)) \\ = & ((r \mapsto r) \mapsto (r \mapsto r)) \end{array}$$

The proposed stratification induces a tree-like organization of typing. This could be generalized to a direct acyclic graph, by extending the endomorphism T to a relation.

To be on the safe side, we assume henceforth that *the typing endomorphism T stratifies objects.*

Validity

In CT0, all compositions of objects are permitted, including those by *rule*. We need to characterize those valid objects which can be recursively deduced from certain objects by certain rule-objects. To this end, we define the **validity** of objects as follows. Given a set of distinguished objects,

- each distinguished object is a valid object;
- if a and b are valid objects, then $a; b$ is a valid object;
- if a is a valid object, then Ta is a valid object;
- there are no other valid objects.

Thus, validity is preserved by *comp* and T, not by *rule*: we may derive $(u \mapsto v); (v \mapsto w) = (u \mapsto w)$ from $u \mapsto v$ and $v \mapsto w$, but we may not throw in any rule-object $t \mapsto s$, even if t and s are valid. *This definition of validity is assumed for all CT-algebras*, with suitable adaptations w.r.t. additional operations.

Note this definition is permissive: it permits rather useless, albeit harmless, objects such as $(a \mapsto b); (c \mapsto d)$ where $b \neq c$, and even where $T^i b \neq T^i c$ for all i. The given definition could be strengthened so as to remove such useless objects.

3.2 Algebras CT1: CT0 plus Application and Product

We enrich CT0 by the operations *app*(a, b) and *prod*(a, b), corresponding to function application and to object product, respectively. We began with composition in CT0 because the composition of deductions seems to be more important than their application.

Signature

The following operations are added to CT0 (§3.1):

$$\begin{array}{rr} \mathit{app},\ \mathit{prod} : \mathit{object} \times \mathit{object} & \longrightarrow \mathit{object} \\ \mathit{sel}_1,\ \mathit{sel}_2 : \qquad\qquad \mathit{object} & \longrightarrow \mathit{object} \end{array}$$

Notations:

$app(a,b)$	$a \setminus b$
$prod(a,b)$	$a \times b$

Axioms

We restate usual properties, e.g. $(a \mapsto b)(a) = a$ and $(c \circ b)(a) = c(b(a))$, and we give the laws ensuring T remains an endomorphism w.r.t. the new operations:

$$a \setminus (a \mapsto b) = b \tag{5}$$
$$a \setminus (b; c) = (a \setminus b) \setminus c \tag{6}$$
$$sel_1(a \times b) = a \tag{7}$$
$$sel_2(a \times b) = b \tag{8}$$

$$T(a \setminus b) = (Ta \setminus Tb) \tag{9}$$
$$T(a \times b) = (Ta \times Tb) \tag{10}$$
$$T(sel_1 a) = sel_1(Ta) \tag{11}$$
$$T(sel_2 a) = sel_2(Ta) \tag{12}$$

Validity is preserved by $app, prod, sel_1, sel_2$.

Illustration

We can now explain more technically why typing is defined as an endomorphism. Let us assume the following context, where $a, b, c, u_i, v_i, w_i (i = 0 \ldots 3)$ are given constants:

$$\begin{aligned} Ta = Tb = Tc &= r \\ Tu_i &= v_i \\ v_0 &= a \\ v_1 &= (a \mapsto b) \\ v_2 &= (b \mapsto c) \\ Tv_3 &= (r \mapsto r) \\ w_i &= Tv_i \end{aligned}$$

We observe

$$level(u_i) = 2, \quad level(v_i) = 1, \quad level(w_i) = 0 \ .$$

At level 2, we may deduce c by d_2, using one lifting by T:

$$\begin{aligned} d_2 &= u_0\backslash(u_1; u_2) \\ Td_2 &= T(u_0\backslash(u_1; u_2)) \\ &= Tu_0\backslash(Tu_1; Tu_2) \\ &= a\backslash((a \mapsto b); (b \mapsto c)) \\ &= c \end{aligned}$$

Using d_1 at level 1, we may again deduce c, and no lifting by T is needed:

$$\begin{aligned} d_1 &= v_0\backslash(v_1; v_2) \\ &= a\backslash((a \mapsto b); (b \mapsto c)) \\ &= c \end{aligned}$$

At level 0, we may only deduce weaker results, e.g. r by d_0:

$$\begin{aligned} d_0 &= w_0\backslash(w_1; w_3) \\ &= r\backslash((r \mapsto r); (r \mapsto r)) \\ &= r \end{aligned}$$

The introduction of d_2 at level 2 is in fact a useless detour "underground", since Td_2 is the real deduction. *The endomorphic nature of T permits the direct deduction d_1 at level 1.* At level 0, the deduction d_0 uses the type of an otherwise unknown rule-object v_3. This yields a sketchy structural information, but is still better than nothing: the result suggests to use some rule-object v_3 of type $Tv_3 = w_3 = (r \mapsto r)$ in order to deduce c which is of type r. Note $v_0\backslash(v_1; v_3)$ of type d_0 cannot be reduced since v_3 is unknown, but it is valid if the v_i are assumed to be valid.

If we want to deduce an object at level k, it is superfluous to use deductions at level $k+1$, viz one level away from the root. If nothing can be deduced at level k, we may try to deduce something less informative at level $k-1$, viz one level closer to the root. We can freely choose the most appropriate abstraction level at which to carry out deductions: the typing endomorphism preserves the structure of deductions and only reduces the semantic richness.

Relationship with categories

CT1-algebras apparently correspond to Cartesian closed categories having an endofunctor; indeed, categories served as a useful source of inspiration because composition is their fundamental constructor. In Cartesian closed categories, morphisms $(m : u \longrightarrow v)$ are internalized as exponential objects v^u; in CT1-algebras, they are rather internalized as objects m the type of which is a rule-object, viz $Tm = (u \mapsto v)$. The typing endomorphism in CT1 corresponds to an endofunctor in the corresponding category.

Results from category theory could be re-used for CT-algebras. For instance, there is a principle of duality: the dual of a CT0-theorem is obtained by transforming $rule(a, b)$

and *comp*(a, b) into *rule*(b, a) and *comp*(b, a); backwards derivations can then defined as the dual of forward derivations.

The choice between CT-algebras and categories is a matter of taste. We now prefer to distinguish between objects according to their type structure, rather than to distinguish between categorical objects and morphisms by definition. However, the present approach could be re-expressed in terms of categories endowed with an endofunctor.

In CT0, one could define an identity-object *id* for composition; this would strengthen the correspondence with categories.

3.3 Algebras CT2: CT1 plus Dependent Product

As well known, dependent products can be naively viewed as infinitary products: we go from binary products (§3.2) to n-ary products and then to ω-ary products. The axioms for products and typing are adapted accordingly. We see a dependent product $\Pi_{x:b}a$ as the product of all objects a^x_c obtained by instantiating the identifier x to an object c such that $Tc = b$. The addition of dependent products is thus an auxiliary matter. The kernel CT0- and CT1-algebras do already provide the gist of the proposed approach.

Signature

The following operations are added to CT1 (§3.2):

$$\begin{aligned} all,\ subst &: (identifier \times object) \times object \to object \\ sel &: \qquad\qquad object \times object \to object \\ T &: (simple \to object) \times object \to object \end{aligned}$$

The identifiers identify place-holders in objects. We assume a one-to-one correspondence between place-holders and identifiers. Hence, distinct binding occurrences always use distinct identifiers. We thereby abstract from issues concerning clashes between concrete identifiers.

The *simple objects* are the constants and the identifiers.

Notations:

$all((x, b), a)$	$\Pi_{x:b}a$
$subst((x, c), a)$	a^x_c
$sel(c, a)$	$sel_c a$
$T(V, a)$	$T_V a$

Axioms

Let V denote a typing valuation from simple objects (i.e. constants and identifiers) to objects; V is total.

$$V : simple \to object\ .$$

A valuation is also known as an environment. In the context of a typing valuation V,

$$sel(c, all((x, T(V, c)), a)) = subst((x, c), a) \ ,$$

viz

$$sel_c(\Pi_{x:T_V c}a) = a_c^x \ .$$

The typing $T_V a$ and the substitution a_c^x preserve the structure of a w.r.t. the operations *rule*, *comp*, *app*, *prod*, sel_1, sel_2, and they verify the following axioms where p is a simple object:

$$T_V p = V p \tag{13}$$
$$T_V \ (\Pi_{x:b} \ a) = \Pi_{x:b} \ (T_{V \cup (x,b)} \ a) \tag{14}$$
$$T_V \ (sel_c \ a) = sel_c \ (T_V \ a) \tag{15}$$

$$x_c^x = c \tag{16}$$
$$p_c^x = p \quad \text{if } p \neq x \tag{17}$$
$$(\Pi_{y:b} \ a)_c^x = \Pi_{y:b_c^x} \ (a_c^x) \tag{18}$$
$$(sel_d \ a)_c^x = sel_{d_c^x} \ (a_c^x) \tag{19}$$

The typing T_V extends the valuation V homomorphically over objects. In (14), T_V modifies neither the type b of the formal parameter x nor the selector c, so as to preserve the structure of ω-ary products: a one-to-one correspondence exists between the components, or instances, of $\Pi_{x:b}a$ and those of $T_V \Pi_{x:b}a$, as it is the case for binary products (10).

Remarks

The application $a \backslash (a \mapsto b)$ of a rule-object $a \mapsto b$ on an object a is in no way related to the selection in a dependent product: we do *not* identify rule application and parameter instantiation.

If $\Pi_{x:b}a$ is valid and $T_V c = b$, then $sel_c(\Pi_{x:b}a)$ is valid. If, for each c such that $T_V c = b$, the object a_c^x is valid, then the object $\Pi_{x:b}a$ is valid. Since in general the latter condition is difficult to prove, sufficient criteria can be used instead; at worst, we may assume that only distinguished dependent products are valid.

The constraint of stratification by typing (§3.1) is easily adapted to the case of dependent products, by excluding recursion in typing-valuations. This can be guaranteed by the following straightforward criterion; to simplify, we assume the root r is the only constant object.

A valuation $V = \bigcup_i (x_i, a_i)$ defines a *strict dependency sequence* iff, for each i, the object a_i uses no other simple object than the root r and the identifiers x_j such that $j < i$. If a typing-valuation defines a strict dependency sequence, then the level of each simple object is finite, and typing does stratify objects. Hence, in a particular language for deductions, identifiers could be used textually before their declarations; only cycles in typing are excluded. The strict left-to-right ordering forces a bottom-up organization of declarations.

Abbreviations could also be introduced, using abbreviation-valuations

$$V_a : simple \rightarrow object \ .$$

To an abbreviation $V_a x = c$ corresponds the dependent object $(x :\overset{d}{=} c)a$ such that

$$(x :\overset{d}{=} c)a = a^x_c \ .$$

If type-valuations V_t and abbreviation-valuations V_a are both provided, the valuations V are composed as

$$V = V_t \cup V_a \ .$$

Since valuations are maps, they could be integrated as first-class objects, viz as products, and then they could also be typed.

3.4 Overall Structure

In order to have a global view of CT2-algebras, we assemble the successive parts of their definition. The hart (§3.1) is given by the axioms on *rule* and *comp*. The binary product *prod* and the selections sel_1, sel_2 (§3.2) are generalized into the dependent product *all* and the selection *sel* (§3.3).

Signature

$$\begin{array}{rr}
\alpha_j : & \rightarrow object \\
rule,\ comp,\ app,\ prod : & object \times object \rightarrow object \\
all,\ subst : & (identifier \times object) \times object \rightarrow object \\
sel_1,\ sel_2 : & object \rightarrow object \\
sel : & object \times object \rightarrow object \\
V : & simple \rightarrow object \\
T : & (simple \rightarrow object) \times object \rightarrow object
\end{array}$$

Notations:

$rule(a,b)$	$a \mapsto b$
$comp(a,b)$	$a;b$
$app(a,b)$	$a \setminus b$
$prod(a,b)$	$a \times b$
$all((x,b),a)$	$\Pi_{x:b}a$
$subst((x,c),a)$	a^x_c
$sel(c,a)$	$sel_c a$
$T(V,a)$	$T_V a$

Axioms for operations

$$\begin{aligned}
(a \mapsto b);(b \mapsto c) &= (a \mapsto c)\\
b \setminus (b \mapsto c) &= c\\
a;(b;c) &= (a;b);c\\
a \setminus (b;c) &= (a \setminus b) \setminus c\\
sel_1(a_1 \times a_2) &= a_1\\
sel_2(a_1 \times a_2) &= a_2\\
sel_c\ (\Pi_{x:T_V c}\ a) &= a_c^x
\end{aligned}$$

Validity is preserved by *comp*, *app*, *prod*, sel_1, sel_2, *sel*, T.

Axioms for typing and substitution

The basic typing valuation V is given. The substitution S homomorphically extends a basic, total substitution S_0 over objects; recall clashes of identifiers are excluded since distinct place-holders are denoted by distinct identifiers.

$$\begin{aligned}
T_V p &= Vp\\
T_V(a \mapsto b) &= (T_V a \mapsto T_V b)\\
T_V(a;b) &= (T_V a; T_V b)\\
T_V(a \setminus b) &= (T_V a \setminus T_V b)\\
T_V(a_1 \times a_2) &= (T_V a_1 \times T_V a_2)\\
T_V\ (\Pi_{x:b}\ a) &= \Pi_{x:b}\ (T_{V \cup (x,b)}\ a)\\
T_V(sel_1 a) &= sel_1(T_V a)\\
T_V(sel_2 a) &= sel_2(T_V a)\\
T_V(sel_c\ a) &= sel_c(T_V a)
\end{aligned}$$

$$\begin{aligned}
Sp &= S_0 p\\
S(a \mapsto b) &= (Sa \mapsto Sb)\\
S(a;b) &= (Sa;Sb)\\
S(a \setminus b) &= (Sa \setminus Sb)\\
S(a_1 \times a_2) &= (Sa_1 \times Sa_2)\\
S(\Pi_{y:b}\ a) &= \Pi_{y:Sb}\ (Sa)\\
S(sel_1 a) &= sel_1(Sa)\\
S(sel_2 a) &= sel_2(Sa)\\
S(sel_d\ a) &= sel_{Sd}\ (Sa)
\end{aligned}$$

4 Related Work

We compare the present approach with related ones, based on variants of typed λ-calculi [3, 4, 11, 13, 14, 20]. Only a few points are stressed; a complete comparative analysis of all relevant approaches, which become rather numerous, is another matter.

4.1 Basic Typed λ-Calculi

In the classical view, typing is not considered as an homomorphism because it is defined by specific inference rules, without equality laws between type expressions; one *directly* infers

$$T(f;g) = (a \mapsto c)$$

from the hypotheses

$$Tf = (a \mapsto b), \;\; Tg = (b \mapsto c) \;\; .$$

Here, we use the endomorphism

$$T(f;g) = (Tf;Tg)$$

plus Axiom (2):

$$(a \mapsto b);(b \mapsto c) = (a \mapsto c) \;\; .$$

The difference boils down to

$$\frac{f : a \mapsto b, \quad g : b \mapsto c}{(f;g) : ((a \mapsto b);(b \mapsto c))} \quad \text{vs.} \quad \frac{f : a \mapsto b, \quad g : b \mapsto c}{(f;g) : a \mapsto c} \;\; .$$

This is like the difference between using computational expressions and writing their results:

$$\frac{f = 3, \quad g = 5}{(f+g) = (3+5)} \quad \text{vs.} \quad \frac{f = 3, \quad g = 5}{(f+g) = 8} \;\; .$$

Recall the objects f and g are in fact superfluous. The deduction of $a \mapsto c$ can remain at the level of $a \mapsto b$ and $b \mapsto c$, viz

$$\frac{a \mapsto b, \quad b \mapsto c}{(a \mapsto b);(b \mapsto c)} \;\; .$$

Another point concerns the style of the definition. The first semantics of classical λ-calculi was operational, viz in terms of reduction rules; higher-level mathematical properties had to be obtained as theorems. Here, algebraic laws by themselves constitute the basic definition, and typing is globally defined as an endomorphism. Reduction rules for expressions can then be derived systematically, as often done for algebraic theories; specific inference rules for typing can be derived similarly.

4.2 Basic λ-Typed λ-Calculi

In such calculi [6, 17], types are viewed as values and can result from the reduction of type-expressions, which are λ-expressions too. Thus, types can be computed at a higher level of abstraction, in the same syntactical framework: typing begins to look like an endomorphism.

However, typing does not preserve the structure of functional composition. Indeed, in λ-typed λ-calculi, simple functional types are still defined by abbreviation, viz

$$(a \mapsto b) \doteq \lambda x : a.b \ .$$

With this definition, we cannot compute at the level of types as we want:

$$(a \mapsto b); (b \mapsto c) \neq (a \mapsto c) \ ,$$

since

$$(\lambda x : a.b); (\lambda y : b.c) \neq (\lambda x : a.c) \ .$$

The semantics of λ-typed λ-calculi is usually given in terms of reduction rules; algebraic laws must be derived as theorems. Moreover, the elaboration of clear mathematical models has proved difficult. This may explain why these calculi are not used more widely.

4.3 Applied λ-Typed λ-Calculi

These applied calculi are elaborations of basic ones in order to improve their expressive power and practical usefulness. For instance, contexts [7] can be introduced, as well as products or iterations.

As in the case of pure λ-typed λ-calculi, typing is not an endomorphism; the semantics remains operational, and is harder to understand because of the additional constructs. The CT-algebras could provide instructive models for such applied calculi, suitably adapted. For instance, we can establish a raw correspondence between CT-algebras and the Deva language, which is one particular version of an applied λ-typed λ-calculus. This correspondence could be organized as follows.

The Deva language

The definition of the Deva language amounts to a technical realization of the naive view of §2 in terms of the Automath-based language Λ and its variants [6, 7, 16, 24]. The platform Λ has been chosen, after a few alternative tentatives, because it is comparatively simple and economical, and because it supports some of the major concepts in §2, especially deductions as objects and types as abstract objects. However, Λ is based on binding and substitution, and does not provide from the start composition, product, and valuations, as needed for our applications in programming. The definition of Deva progressively integrated these concepts on top of Λ. They were first introduced as definitional extensions. Then, they were gradually shifted into the language kernel, with parallel adaptations [8, 9, 27] of the normalization proofs established for Λ [16, 24]. The resulting language has been published essentially through case-studies [1, 10, 12, 26]: this is none-too-surprising since case-studies provided important and continued guidance.

Algebraic view of Deva

We present this view as a set of straightforward correspondences between CT2 and Deva. Some of these correspondences are approximative, for reasons detailed below. We use braces to indicate which concepts have not yet been integrated on one side or the other:

In CT2:	*In Deva:*
objects	texts
root r	**prim**
rule $a \mapsto b$	{rule}
composition $a; b$	cut $a \circ b$
application $a \setminus b$	{modus ponens}
product $a_1 \times a_2$	product $[a_1, a_2]$
dependent product $\Pi_{x:b}a$	abstraction $[x : b \vdash a]$
selection $sel_c a$	application $c \setminus a$
{coproduct $a_1 + a_2$}	sum $[a_1 \| a_2]$
typing T	typing Typ
type-valuation (x, a)	declaration $x : a$
type-constraint $Ta = b$	judgement $a \therefore b$
{abbreviation (x, c)}	definition $x := c$
{valuations}	contexts
{operations on valuations}	operations on contexts
{recursion}	iteration
{object synthesis}	implicit texts

The Deva typing *Typ* differs from the CT-typing T essentially in the case of rule-objects:

$$T(a \mapsto b) = Ta \mapsto Tb$$
$$Typ(a \mapsto b) = \ a \mapsto Typ(b)$$

Typ does not lift the left-hand-side of rule-objects, and is thus not completely homomorphic; see §4.2. To obtain an homomorphism, one would need to distinguish rule-application from projection (viz selection or instantiation), i.e. to distinguish a rule $[a \vdash b]$ from a mere abbreviation of the abstraction $[x : a \vdash b]$.

The above correspondence between CT2 and Deva illustrates how structural views influenced the design of the Deva language, if not the details of its definition. The technical description (e.g. [27]) has been based on the operational semantics of basic λ-typed λ-calculi. It would be useful to base the definition on algebraic properties. One would thereby improve the general understanding of type-based approaches, especially w.r.t.

hierarchies of types; the proposed endomorphic view of typing is an example of possible clarification. Moreover, algebraic models can serve as a scientific yardstick for comparing alternative frameworks, and thus as a helpful safeguard against the Babel-tower syndrome.

A interesting phenomenon occurs when program derivations are formalized in Deva. One tends to highlight deductions directly at the level of rules, rather than at the level of objects typed by rules; the adequate level of thinking is indeed that of rules. The "judgement" construction is introduced for that reason: it permits to pinpoint intermediate type-results, if they are important for understanding the deductions; the use of objects typed by rules is in fact a detour.

The Deva-typing is often presented informally as a "quasi"-homomorphism. Yet, the actual deductions in Deva remain blocked at the underground level (§4.2); hence, the semantics of the "cut" operator, which is akin to composition in CT, must be given in terms of complex λ-combinations rather than by Axiom (2).

Further work

Specific variants of CT-algebras can be defined by introducing stratification of typing, restrictions on composition, and results from category theory. One such result could be the generalization of Axiom (2) to products of rules:

$$\Pi_{x:d}(a \mapsto b); \Pi_{x:d}(b \mapsto c) = \Pi_{x:d}(a \mapsto c) \ .$$

This would permit the structured creation of new valid parametrized rules.

As noted in §3.2, the algebraic definition of operations on valuations, e.g. composition, product, and importation, remains to be worked out. Total iterations could be included. It would also be useful to include some of the so-called "implicit" Deva computations, e.g. partial iterations, object synthesis, non-determinism, and higher-order unifications; such computations in fact provide basic means of automated assistance (§1.4). This would certainly help users, but would entail a significant adaptation of the definition of CT-algebras. It is unclear where to draw the line.

5 Conclusion

We propose to compose deductions at the level of the basic deduction-rules, and to classify such deductions by a typing endomorphism. The results of typing are again deductions which may use the same operations, but which provide less semantic information. Thus, types are viewed as specific "values" to be substituted for identifiers, and type derivation amounts to a computation with these "values". The typing computation can itself be defined by a computation expression at a further level of abstraction, and this new expression in turn can be typed. As a matter of fact, the most frequent deductions are those done with classifier objects, viz with types: these are the really ordinary deductions. In deductions, we prefer to work with rules as such, rather than with underground objects typed by these rules. Hence, the "special" objects are those at the leaves of the classification tree, viz the usual ordinary values. This is just a change of perspective.

Acknowledgements. The author gratefully acknowledges questions, suggestions, and ideas from M. Weber, Ph. de Groote, B. Tubiermont, M. Proietti, participants at the IFIP WG2.1 State-of-the-Art Seminar (Brazil, 1992), and members of IFIP WG2.3.

References

1. Bert D. and S. Sebbar, Synthesizing abstract data type representation in the Deva meta-calculus, in [15] 427-450.
2. Broy M. and C.B. Jones (eds.), *Programming Concepts and Methods*, North-Holland, Amsterdam, 1990.
3. Constable R. L., Type theory as a foundation for computer science, in: T. Ito and A.R. Meyer (eds.), *Theoretical Aspects of Computer Software*, LNCS 526, Springer, Berlin, 1991, 226-243.
4. Coquand Th., Metamathematical investigations of a calculus of constructions, in: P. Odifreddi (ed.), *Logic and Computer Science*, APIC SDP 31, Academic Press, London, 1990, 91-122.
5. Curry H.B. and R.Feys, *Combinatory Logic*, vol.I, North-Holland, Amsterdam, 1958.
6. de Bruijn N.G., Generalizing Automath by means of a lambda-typed lambda-calculus, in: *Mathematical Logic and Theoretical Computer Science*, LNM 106, Marcel Dekker, New York, 1987, 71-92.
7. de Bruijn N.G., A plea for weaker frameworks, in: [11] 40-68.
8. de Groote, Ph., *Définition et Propriétés d'un Métacalcul de Représentation de Théories*, Ph. D. Thesis, Univ. of Louvain, 1990.
9. de Groote Ph., Nedepelt's calculus extended with a notion of context as a logical framework, in: [11] 69-86.
10. Gabriel R., Program transformation expressed in the Deva meta-calculus, in: [15] 267-287.
11. Huet G. and G. Plotkin (eds.), *Logical Frameworks*, Cambridge U. Press, Cambridge, 1991.
12. Lafontaine C., Formalization of the VDM reification in the Deva meta-calculus, in [2] 333-368.
13. Lambek J.L. and P.J. Scott, *Introduction to Higher Order Categorical Logic*, CSAM 7, Cambridge U. Press, Cambridge, 1989.
14. Mitchell J.C., Type systems for programming languages, in: J. van Leeuwen (ed.), *Formal Models and Semantics*, Handbook Theor. Comp. Sci. B, Elsevier, Amsterdam, 1990, 365-458.
15. Möller B. (ed.), *Constructing Programs from Specifications*, North-Holland, Amsterdam, 1991.
16. Nederpelt R.P., *Strong normalization in a typed lambda-calculus with lambda structured types*, Ph.D. Thesis, Techn. Univ. Eindhoven, 1973.
17. Nederpelt R.P., An approach to theorem proving on the basis of a typed lambda-calculus, in: W. Bibel and R. Kowalski (eds.), *5th Conference on Automated Deduction*, LNCS 87, Springer, 1980, 182-194.
18. Pepper P., A simple calculus for program transformation (inclusive of induction), *Sci. Comput. Programming*, 9(1987) 221-262.
19. Pitt D. et al. (eds.), *Category Theory and Computer Programming*, LNCS 240, Springer, Berlin, 1986.
20. Poigné A., Category theory and logic, in [19] 103-142.
21. Sintzoff M., Suggestions for composing and specifying program design decisions, in: B. Robinet (ed.), *Proc. 4th Symp. on Programming*, LNCS 83, Springer, Berlin, 1980, 311-326.

22. Sintzoff M., Understanding and expressing software construction, in: P. Pepper (ed.), *Program Transformations and Programming Environments*, ASI F8, Springer, Berlin, 1984, 169-180.
23. Sintzoff M., Expressing program development in a design calculus, in: Broy M. (ed.), *Logic of Programming and Calculi of Discrete Design*, ASI F36, Springer, Berlin, 1987, 343-365.
24. van Daalen D.T., *The Language Theory of Automath*, Ph.D. Thesis, Techn. Univ. Eindhoven, 1980.
25. van Gasteren A.J.M., *On the Shape of Mathematical Arguments*, LNCS 445, Springer, Berlin, 1990.
26. Weber M., Formalization of the Bird-Meertens algorithmic calculus in the Deva meta-calculus, in: [2] 201-232.
27. Weber M., *A Meta-Calculus for Formal System Development*. Ph. D. Thesis, Univ. Karlsruhe, 1990, and: R. Oldenbourg Verlag, München, 1991.