Algorithmic Languages, de Bakker/van Vliet (eds.)

On the Notion of Strong Typing

Maarten M. Fokkinga

Twente University of Technology, P.O. Box 217, 7500 AE Enschede, The Netherlands

The usefulness of strong typing is formalized in the following way. Strong typing is a syntactic means to restrict the class of programs so that a pleasant semantic property holds. More precisely, a semantic equivalence of strongly typed programs is proved independent of the representation used to implement abstract entities like numbers, truth values and predefined ones.

Thus a formal content is given to phrases like "typing prevents to employ unintended properties of representations" and "semantically types are redundant".

1. Introduction

It seems widely accepted that so-called strong typing has some undeniable benefits. E.g. the ALGOL 68 designers claim that "ALGOL 68 has been designed in such a way that most syntactic errors and many others can be detected easily before they lead to calamitous results" [19, Section 0.1.3]. Undoubtly it is its mode discipline which plays a major role in this error detection (see [6, 8]). Indeed, "one often pays a price for [the absence of a type system] in the time taken to find rather inscrutable bugs – anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms" [11].

It is therefore not surprising that the following requirement is included in STEELMAN [2]:

> "3A. Strong Typing. The language shall be strongly typed. The type of each variable, array, record, expression, function and parameter shall be determinable during translation".

But STEELMAN neither provides a formal definition of strong typing, nor does it give any semantic property aimed at in requiring strong typing. So how could one prove that ADA meets the requirements or desiderata?

In this paper we investigate what formally the usefulness of strong typing might be. To this end we view typing as a purely syntactic way of restricting the class of programs so that a pleasant semantic property holds for that class, and we thus formalize the interplay between the syntactic typing and the semantic properties of programs. This view is in accordance with [13] and [12], and seems consistent with practical implementations of strongly typed languages. Nevertheless one mostly finds types motivated in a setting where semantic entities (like retracts [16] and [3], downward closed directed c.p.o.'s [11] and so on) are assigned to types.

Our paper might be viewed as a continuation of [15] and [3]. They both present a theorem which we call the Correspondence Theorem. Informally this theorem asserts that there is a relation, called correspondence, which relates for any strongly typed program the values denoted under different implementations. However, both assign a semantics to types. We are glad to improve their results in that we show types to be semantically redundant. Moreover we prove a nicer theorem (Theorem 3.10) which asserts that a *semantic equivalence of programs* is independent of the implementation.

The formalization and proofs are carried out in the framework of the typed λ-notation. We define two expressions equivalent with respect to some type t if their values, when used according to t, are the same function – or constant.

The remainder of the paper is organized as follows. In Section 2, we formally define syntactic concepts of the language, and define some axioms which are to characterize the semantics. In Section 3, the formalization of the usefulness of strong typing is presented. Thereafter, in Section 4, we give a specific semantics of the language, satisfying the axioms; that section only serves to provide a concrete example. Finally we conclude with Section 5, discussing the results obtained.

2. The Language

We choose a simple language to illustrate the essential ideas. Obviously, then, the language has to have a construct where type checking is involved, say function application or assignment. Moreover the language has to have a construct for user controlled creation of new values; were this not the case there would be no problems at all, because one must of course assume that all 'predefined' values behave well. In view of its simple semantics we are led to consider the λ-notation; λ-abstraction is the construct to create new values.

Definition 2.1 (Expressions and Types). Let X be a countably infinite set of *normal identifiers* and let Z be a set of *type identifiers*. Throughout the paper we let x and y vary over X and z over Z. Specific elements of X are e.g.

$$zero,\ one,\ succ,\ pred,\ true,\ false, \ldots$$

and specific elements of Z are

$$int,\ bool, \ldots .$$

The set T of *types* is defined thus

$$t ::= z \mid (t \to t').$$

The set E of *expressions* is defined thus

$$e ::= x \mid (\lambda x : t \,.\, e) \mid e(e').$$

Throughout the paper we let e vary over E and t over T; we sometimes suffix them with digits, primes and letters f, a and b (for *f*unction, *a*rgument and *b*ody). According to common usage we omit parentheses when they are clear from the context; in particular the scope of λ extends as far as possible, and $\to$ associates to the right, so that $t1 \to t2 \to t3 = t1 \to (t2 \to t3)$.

Notice that there are no constants like **0, 1, 2, ...** ; predefined identifiers like *zero, one, two, ...* (or even *zero* and *succ* alone) should enable the programmer to use numbers. Other interesting predefined identifiers may be the so-called fixed point operators, $fixpoint_{t,t'}$ of type $((t \to t') \to (t \to t')) \to (t \to t')$, to enable recursive definitions.

Syntactic sugar might be added to make the language more practical. E.g. non-recursive definitions can be introduced as an abbreviation:

$$\textbf{let } x : t = e' \textbf{ in } e$$

and

$$e \textbf{ where } x : t = e'$$

abbreviate $(\lambda x:t\,.\,e)(e')$. Also conditional expressions can be introduced:

$$\textbf{if } e \textbf{ then } e1 \textbf{ else } e2$$

where both $e1$ and $e2$ have type t, abbreviates

$$cond_t(e)(\lambda x:null\,.\,e1)(\lambda x:null\,.\,e2)$$

where $cond_t$ has type $bool\to(null\to t)\to(null\to t)\to t$. All this is well known, see e.g. [17].

We now define what expressions are well typed. The formal term used is strong typing. Informally it means that for each application the type of the argument must match the parameter type of the function. In our simple language two types match iff they are equal; in a more elaborate language a less trivial relation may hold.

The type of identifiers depends on the context in which they occur. We model that context by a so-called syntactic environment. Formally, the set S of *syntactic environments* is the set of partial functions $X\to T$. Throughout the paper we let s vary over S. For each s we assume that there exists an identifier x which has not yet a type associated with it; we say that $new(x,s)$ holds in that case. In view of the infinity of X this is no strong requirement.

As usual the suffix $[p\leftarrow q]$ denotes *updating of a function;* in particular

$$s[x\leftarrow t](x') = \text{if } x=x' \text{ then } t \text{ else } s(x').$$

This notation will also be used for semantic environments r, to be introduced below.

Definition 2.2 (Strong Typing). The relation $s\vdash e:t$ (*"e has type t in s"*) is the smallest relation satisfying

(a) if $s(x)=t$, then $s\vdash x:t$;

(b) if $s[x\leftarrow ta]\vdash eb:tb$, then $s\vdash(\lambda x:ta\,.\,eb):(ta\to tb)$;

(c) if for some ta, $s\vdash ef:ta\to tb$ and $s\vdash ea:ta$, then $s\vdash ef(ea):tb$.

We say *e is strongly typed in s* if for some t, $s\vdash e:t$.

Now we turn to the semantics of the language. Let V be the set of *values* which serve as meanings for expressions, and let $R=X\to V$ be the set of *semantic environments* giving the meaning of the predefined identifiers. (Throughout we let v and w vary over V and r over R.) The *meaning of an expression e* is then given by $M(e,r)$, where $M\in E\times R\to V$ is the so-called meaning function (a partial function).

Usually the meaning of expressions are taken to be some *abstract entities,* like numbers, truth values or functions. Accordingly the domain of numbers is associated with the type identifier *int,* the domain of truth values is associated to *bool,* and – sometimes mathematically quite sophisticated – functional domains are associated to types $t \to t'$. Actually, however, expressions yield bit patterns, or the like, which in some way or another represent those abstract entities. And accordingly, from the bit pattern alone, say *concrete value,* one can not tell whether it is meant as a number, truth value or function. It is indeed quite possible to execute a bit pattern meant as a number as if it represents a function. Thus semantically types do not enter the picture.

Admittedly, mostly the abstract entities are of interest. But the interpretation of the concrete values cannot be the task of the language designer, i.e. is not incorporated into M. Even if M would produce numbers, then still these numbers represent some more abstract entities like year of birth, salary and so on. The interpretation is really outside the grip of M, and is left to the individual programmer and creator of the standard environment.

Consequently the value denoted by an expression is possibly untyped. We will however not burden the reader/programmer with details of the value space V, but instead specify the meaning of expressions by the axioms which we need in the proofs below.

Definition 2.3 (Axioms for M). For strongly typed expressions the meaning function satisfies the following axioms.

(a) $M(x,r)=r(x)$;

(b) if $v=M(ea,r)$, then $M((\lambda x:ta\,.\,eb)(ea),r)=M(eb,r[x\leftarrow v])$;

(c) if y not free in e, then $M((\lambda x:ta\,.\,e)(ea),r)=M((\lambda y:ta\,.\,e[x/y])(ea),r)$;

(d) if $v=M(e',r)$ and x does not occur free in the scope of some λ within e, then $M(e[x/e'],r)=M(e,r[x\leftarrow v])$.

Above, and in the sequel, we use the postfix $[x/e']$ to denote *substitution* of e' for x – taking care to rename bound identifiers in order to avoid clash of names.

Notice that $M(\lambda x:t\,.\,e,r)$ need not be a function. All we require is that it can be used as a function in the sense of axioms (b) and (c). Indeed, the M given in Section 4 will yield some code of a function, so that e.g. $M(\lambda x:t\,.\,e,r)$ differs from $M(\lambda y:t\,.\,e[x/y],r)$. Actually in Section 4 we take

V to be a set of untyped values, so that any value may be used in any way, and M even satisfies the axioms for not strongly typed expressions.

In the sequel we will use the following abbreviations.

(1) '$v(w)$' abbreviates $M(x(y), r[x \leftarrow v, y \leftarrow w])$, and is thus a concise way of expressing that v is to be used as a function with argument w.

(2) $e1 =_r e2$ abbreviates $M(e1, r) = M(e2, r)$; $e1$ and $e2$ yield the same value in r.

3. Formalizing the Usefulness of Strong Typing

We will first introduce the syntactic concept of primitive expressions. These denote what one might call predefined values and they are used to state assumptions on alternative representations for the same set of abstract entities. Secondly we define the semantic relation of correspondence and some properties of it. The correspondence relation is used in the proof of Theorem 3.10 which expresses our view on the usefulness of strong typing.

Suppose that the standard environment r provides via $zero : int$ and $succ : int \rightarrow int$ an implementation for numbers. Of course, the concrete value denoted by $zero$ is not the number zero, but merely represents it in some way or another. We may also consider an alternative implementation $\hat{r}$. Surely $r(zero)$ and $\hat{r}(zero)$ need not be equal, although they both represent the same abstract entity. E.g. the expressions

$$zero,\ succ(zero),\ succ(succ(zero)), \ldots$$

constitute the – unknown – representation of numbers. And if e.g. $pred : int \rightarrow int$ is also present, then

$$pred(zero),\ pred(succ(zero)),\ succ(pred(zero)), \ldots$$

might also contribute to the representation. However note that abstractions like $\lambda x : int . x$ or $\lambda x : int . zero$ do not contribute to the representation of abstract entities as far as determined by the environment. Thus we are led to the following definition to get some grip on the representations of abstract entities.

Definition 3.1 (Primitive expressions). For any s the set P of *primitive expressions* consists of all strongly typed expressions p generable by

$$p ::= x \mid p(p).$$

In the sequel p varies over P.

It will turn out that, for fixed s and r, the primitive expressions of type z constitute all expressible 'z-values'. Thus they play the role usually played by constants. However we do not restrict the types of the given function identifiers to first order; a function identifier $mk\text{-}int:((t \to t) \to int)$ may occur in the primitive expressions and so contribute to the values representing 'int's.

Given two environments r and $\hat{r}$, we wish to define a correspondence relation $\sim_t$ on $V \times V$, relating those values which *wrt* r resp $\hat{r}$ represent the same abstract entity. As one concrete value may represent a variety of abstract entities (e.g. 001 may represent both the number one and the truth value true, and many more), we need to indicate with respect to what interpretation the correspondence is to be understood. The type t serves that purpose. Of course we want $M(p, r) \sim_z M(p, \hat{r})$ for p of elementary type z; thus the relation also depends on s.

Definition 3.2 (Correspondence). For any s, r, $\hat{r}$ and t the relation $s, r, \hat{r} \vdash v \sim_t \hat{v}$ (*"v and $\hat{v}$ represent the same abstract entity"*) is defined by induction on t as follows:

(a) $t = z$: $s, r, \hat{r} \vdash M(p, r) \sim_z M(p, \hat{r})$ for any p with $s \vdash p : z$;

(b) $t = ta \to tb$: $s, r, \hat{e} \vdash v \sim_t \hat{v}$ iff for all w, $\hat{w}$ with $s, r, \hat{r} \vdash w \sim_{ta} \hat{w}$, also $s, r, \hat{r} \vdash$ '$v(w)$' $\sim_{tb}$ '$\hat{v}(\hat{w})$'.

We cannot expect to derive any interesting property for the correspondence relation unless we assume consistency between the two environments. In particular the following predicate $Correct\sim(s, r, \hat{r})$ is reasonable.

Definition 3.3 (Correct$\sim$). $Correct\sim(s, r, \hat{r})$ holds iff for all x, t with $s \vdash x : t$

$$s, r, \hat{r} \vdash M(x, r) \sim_t M(x, \hat{r}).$$

The following lemma shows that a seemingly stronger requirement for $Correct\sim(s, r, \hat{r})$ actually already follows from the given definition.

Lemma 3.4. *Let s, r, $\hat{r}$ satisfy $Correct\sim(s, r, \hat{r})$. Then, for any p, t with*

$s \vdash p : t$,

$$s, r, \hat{r} \vdash M(p, r) \sim_t M(p, \hat{r}).$$

Proof. By induction on the structure of p.

The following lemma is needed to prove the Stability of Correspondence Lemma below, which in turn is needed in the Correspondence Theorem following it. Both lemmata are of a rather technical nature. They show that updating of s, r, $\hat{r}$ to $s[x \leftarrow t]$, $r[x \leftarrow v]$, $\hat{r}[x \leftarrow \hat{v}]$ under certain circumstances does not change the relation $\sim_t$.

Lemma 3.5. *Let s, r, $\hat{r}$ satisfy Correct$\sim(s, r, \hat{r})$; let w, $\hat{w}$, ty satisfy $s, r, \hat{r} \vdash w \sim_{ty} \hat{w}$; let y be new in s, i.e. new(y, s). Then for any p, t with $s' \vdash p : t$,*

$$s, r, \hat{r} \vdash M(p, r') \sim_t M(p, \hat{r}')$$

where $s' = s[y \leftarrow ty]$, $r' = r[y \leftarrow w]$, $\hat{r}' = \hat{r}[y \leftarrow \hat{w}]$.

Proof. By induction on the structure of p.

Lemma 3.6 (Stability of Correspondence). *Let s, r, $\hat{r}$ satisfy Correct$\sim$ $(s, r, \hat{r})$; let w, $\hat{w}$, ty satisfy $s, r, \hat{r} \vdash w \sim_{ty} \hat{w}$; let y be new in s, new(y, s). Then for any v, $\hat{v}$, t*

$$s, r, \hat{r} \vdash v \sim_t \hat{v} \quad \textit{iff} \quad s', r', \hat{r}' \vdash v \sim_t \hat{v}$$

where $s' = s[y \leftarrow ty]$, $r' = r[y \leftarrow w]$, $\hat{r}' = \hat{r}[y \leftarrow \hat{w}]$.

Proof. By induction on the structure of t.

Case $t = z$, $\Rightarrow$. Assume $s, r, \hat{r} \vdash v \sim_z \hat{v}$. By definition, for some p with $s \vdash p : z$, $v = M(p, r)$ and $\hat{v} = M(p, \hat{r})$. Because new(y, s), y does not occur free in p, hence $v = M(p, r')$ and $\hat{v} = M(p, \hat{r}')$ and $s' \vdash p : z$. So by definition $s', r', \hat{r}' \vdash v \sim_z \hat{v}$.

Case $t = z$, $\Leftarrow$. Apply Lemma 3.5.

Case $t = ta \rightarrow tb$. Use the definition of correspondence and the induction hypotheses for both ta and tb.

Theorem 3.7 (Correspondence). *Let s, r, $\hat{r}$ satisfy Correct$\sim(s, r, \hat{r})$. Then for any e, t with $s \vdash e : t$*

$$s, r, \hat{r} \vdash M(e, r) \sim_t M(e, \hat{r}).$$

Proof. By induction on the structure of e.

Case $e = x$. Immediate from the assumption.

Case $e = ef(ea)$. Straightforward by induction.

Case $e = \lambda x{:}ta.eb$. Then for some tb, $t = ta \to tb$ and $s[x \leftarrow ta] \vdash eb{:}tb$. Now let w, $\hat{w}$ be arbitrary satisfying $s, r, \hat{r} \vdash w \sim_{ta} \hat{w}$. One may easily verify that '$M(e,r)(w)$' = '$M(\lambda x{:}\,ta\,.\,eb, r)(w)$' $= M(eb[x/y], r[y \leftarrow w])$ where y is chosen such that $new(y,s)$. Setting $s' = s[y \leftarrow ta]$, $r' = r[y \leftarrow w]$ and $\hat{r}' = \hat{r}[y \leftarrow \hat{w}]$, we can show $Correct\sim(s', r', \hat{r}')$ from the Stability of Correspondence Lemma. Hence we may apply the induction hypothesis and find

$$s', r', \hat{r}' \vdash M(eb[x/y], r') \sim_{tb} M(eb[x/y], \hat{r}').$$

As above $M(eb[x/y], \hat{r}') =$ '$M(e, \hat{r})(\hat{w})$', so that

$$s', r', \hat{r}' \vdash \text{'}M(e,r)(w)\text{'} \sim_{tb} \text{'}M(e,\hat{r})(\hat{w})\text{'}.$$

Using once more the Stability of Correspondence Lemma we find

$$s, r, \hat{r} \vdash \text{'}M(e,r)(w)\text{'} \sim_{tb} \text{'}M(e,\hat{r})(\hat{w})\text{'}.$$

We conclude therefore $s, r, \hat{r} \vdash M(e,r) \sim_t M(e,\hat{r})$.

Reynolds [15] and Donahue [3] give more or less this theorem as the effect strong typing has on the semantics of expressions. One may interpret the theorem that an implementor of the predefined values, accessible via the predefined identifiers, may freely switch from one representation r to another $\hat{r}$, provided $Correct\sim(s, r, \hat{r})$, without essentially affecting the value denoted by an expression: the two values do correspond and therefore do represent the same abstract entity; in particular if the expression has a non-composite type we know that the two values $M(e,r)$ and $M(e,\hat{r})$ arise from the same primitive expression.

Yet we feel a bit unhappy with this result; it involves too much hand waving to convince an unwilling listener of the importance. Fortunately there is a more appealing semantic property of strongly typed expressions. Switching from one representation to another does not affect the meaning of expressions in the sense that *semantic equivalence* is unaffected. Semantic equivalence need be defined precisely, because there are several reasonable choices, which in general do not coincide (see e.g. [1]). We choose the one in which two expressions e and e' are said equivalent with respect to a type $t1 \to t2 \to \cdots \to tn \to z$ if there is no context of the form $[\cdots](e1)(e2)\cdots(en)$ with $e1 : t1, \ldots, en : tn$ which discriminates between e

and e'; i.e. $e(e1)(e2)\cdots(en)$ and $e'(e1)(e2)\cdots(en)$ yield the same value. Formally, we define this notion by induction on t.

Definition 3.8 (Equivalence). For any s, r, $e1$, $e2$ we define $s,r\vdash e1\simeq_t e2$ ("*e1 and e2 are equivalent wrt t*") as follows.

(a) for $t=z$: $s,r\vdash e1\simeq_z e2$ if $e1=_r e2$;

(b) for $t=ta\leftarrow tb$: $s,r\vdash e1\simeq_t e2$ if for all e with $s\vdash e: ta$, $s,r\vdash e1(e)\simeq_{tb}$ $e2(e)$.

Notice that $s,r\vdash e1\simeq_t e2$ in itself does not require that $s\vdash e1,e2:t$. Hence it makes sense to consider the question whether any $e1$ and $e2$ are equivalent. In particular we may consider expressions which are not strongly typed, but are *weakly typed* according to [3]. Some simple examples are treated after Theorem 3.10.

An alternative notion of equivalence is the following. Two expressions $e1$ and $e2$ are said equivalent *wrt* type t if there is no strongly typed context $C[\cdots]$ with a hole of type t and as a whole of type z, for some z, such that $C[e1]$ and $C[e2]$ have different values; cf. [10]. Our Theorem 3.10 fails for this notion because of possible pathological values for higher order function identifiers. We might exclude such values by suitable assumptions about r, but we will not pursue this alternative here.

We can of course not expect to prove that equivalence is independent of the environment, unless we assume some consistency requirements between the environments under consideration. In particular the following predicate $Correct\simeq(s,r,\hat{r})$ seems reasonable.

Definition 3.9 (Correct $\simeq$). $Correct\simeq(s,r,\hat{r})$ holds iff for all $p1$, $p2$, z with $s\vdash p1,p2:z$

$$s,r\vdash p1\simeq_z p2 \quad \textit{iff} \quad s,\hat{r}\vdash p1\simeq_z p2.$$

Theorem 3.10 (Representational Independence of Equivalence). *Let s, r, $\hat{r}$ satisfy $Correct\sim(s,r,\hat{r})$ and $Correct\simeq(s,r,\hat{r})$. Then for any $e1$, $e2$, t with $s\vdash e1,e2:t$*

$$s,r\vdash e1\simeq_t e2 \quad \textit{iff} \quad s,\hat{r}\vdash e1\simeq_t e2.$$

Proof. By induction on t.

Case $t = z$, $\Rightarrow$. From $s, r \vdash e1 \simeq_z e2$ we find $e1 =_r e2$ (1)

From $s, r, \hat{r} \vdash M(e1, r) \sim_z M(e1, \hat{r})$ (by the Correspondence Theorem) and similarly for $e2$, we find by the Correspondence Definition

for some $p1$ with $s \vdash p1 : z$, $e1 =_r p1$ and $p1 =_{\hat{r}} e1$,

for some $p2$ with $s \vdash p2 : z$, $e2 =_r p2$ and $p2 =_{\hat{r}} e2$.

Hence by (1) $p1 =_r p2$, so $s, r \vdash p1 \simeq_z p2$, so by $Correct\simeq(s, r, \hat{r})$ also $s, \hat{r} \vdash p1 \simeq_z p2$, so $p1 =_{\hat{r}} p2$ and hence $e1 =_{\hat{r}} e2$, i.e. $s, \hat{r} \vdash e1 \simeq_z e2$.

Case $t = z$, $\Leftarrow$. Similar.

Case $t = ta \rightarrow tb$. Easy by induction.

It is not difficult to construct counter examples to the conclusion of the theorem in case the condition $s \vdash e1, e2 : t$ is not met. E.g. consider the syntactic environment with $zero : int$ and $true, false : bool$. Now let the representation of booleans be a subset of the representation of the integers. In particular choose r and $\hat{r}$ such that

$$r(zero) = r(true) \neq r(false),$$

$$\hat{r}(zero) = \hat{r}(false) \neq \hat{r}(true).$$

Clearly

$$s, \hat{r} \vdash zero \not\simeq_t true \quad \text{for all noncomposite } t \in Z,$$

but yet

$$s, r \vdash zero \simeq_t true \quad \text{for all } t.$$

Donahue [3] defines a notion of weak typing so that $e = (\lambda x : bool . x)(zero)$ is weakly typed and has type *bool*. Again we find $s, r \vdash e \simeq_{bool} true$ but $s, \hat{r} \vdash e \not\simeq_{bool} true$. Thus relaxing the requirement $s \vdash e1, e2 : t$ in the theorem to "$e1$ and $e2$ must be weakly typed, with type t say, in s" invalidates the conclusion.

4. A concrete semantics for the language

This section only serves to show that untyped values and coinciding representations are quite reasonable. We will work out the set V and function M, without any sophisticated mathematical constructions as commonly used in the field of denotational semantics, cf. [3, 9, 15, 16].

Our starting point is that values are untyped, like bit patterns, and that each value may be used in any way. This is just the opposite of Definition 2.1.1.2.c of the ALGOL 68 Report [19], and of the postulation by [5]. For ease of presentation we choose a set V which suits our purpose very well.

Definition 4.1 (The value space V). Let C be a fixed set of *constants,* disjoint from X. The set U of *pseudo-values* is defined by BNF:

$$u ::= x \mid (\lambda x.\, u) \mid u(u') \mid c.$$

The set V of *values* is defined thus

$$V = \{u \in U \mid \text{no } x \in X \text{ occurs free in } u\}.$$

Throughout v and w vary over V; specific elements of C are $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{S}, \mathbf{P}, \ldots$.

Values may be thought to model states of a machine. Possible state transitions are modelled by transformation or reduction rules. A completed transformation of some initial state v into a final state is called the elaboration of v. We choose here a deterministic transformation in applicative order ('call by value'), cf. the SECD machine of [7].

Definition 4.2 (Transformation rules and Elaboration). The deterministic *transformation* $v \to w$ is defined thus:

(a) if $v \to v'$, then $v(w) \to v'(w)$;
(b) if $\forall v'.\, v \not\to v'$ and $w \to w'$, then $v(w) \to v(w')$;
(c) if $\forall w'.\, w \not\to w'$, then $(\lambda x.\, v)(w) \to v[x/w]$;
(d) for each $c \in C$ there is a fixed set of rules

$$c(v1)(v2) \cdots (vn) \to w$$

which respects the deterministic applicative order.

The elaboration $elab \in V \to V$ (a partial function) is given by

$$elab(v) = w \quad \text{if } v \xrightarrow{*} w \text{ and } \forall w'.\, w \not\to w'.$$

In the above framework "fatal errors during elaboration" may be modelled by nontermination. To this end let $\mathbf{error} \in C$ with $\mathbf{error} \to \mathbf{error}$.

Abstract entities like natural numbers N or truth values may be represented in V in a variety of ways, as shown in the next example.

Example 4.3 (Representations of natural numbers). One way is to let $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots \in C$ and to represent $n \in N$ by the obvious constant, say $\mathbf{c}_n$. Further, let $\mathbf{S}, \mathbf{P} \in C$ represent the successor and predecessor function. The following rules are needed: for all n

$$\mathbf{S}(\mathbf{c}_n) \to \mathbf{c}_{n+1},$$

$$\mathbf{P}(\mathbf{c}_{n+1}) \to \mathbf{c}_n.$$

Alternatively, we may represent n by

$$(\lambda x . \lambda y . x^n(y)) = \lambda x . \lambda y . x(x(\cdots x(y) \cdots)),$$

and the successor by $\lambda x . \lambda y . \lambda z . y(x(y)(z))$ and the predecessor either by

$$\mathbf{P} \in C \text{ with } \mathbf{P}(\lambda x . \lambda y . x^{n+1}(y)) \to \lambda x . \lambda y . x^n(y)$$

or by

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

from [17]. There are various other representations with constant-free values, and which have a lower elaboration complexity (see [14]).

In particular the last representation in the above example shows that values are untyped. $\lambda x . \lambda y . y$ represents the number zero, but it may be applied to any value. In fact it also represents any function $f \in A \to B \to B$ with

$$f(a) = \text{identity function on } B.$$

Finally we define M. The role of types is to single out the strongly typed expressions, i.e., those for which Theorems 3.7 and 3.10 hold. Semantically "types are redundant."

Definition 4.4 (The meaning function M). The *compilation* $\bar{\ } \in E \to U$ is defined thus (it throws away all types):

(a) $\bar{x} = x$,
(b) $\overline{(\lambda x : t . e)} = (\lambda x . \bar{e})$,
(c) $\overline{e(e')} = \bar{e}(\overline{e'})$.

The meaning function $M \in E \times R \to V$ is defined

$$M(e, r) = elab(\bar{e}[x/r(x), \text{ for each } x \text{ free in } \bar{e}]).$$

It should be easy to verify the axioms assumed in Definition 2.3, and to construct suitable values for the identifiers $fixpoint_{t,t'}$ and $cond_t$ mentioned in Section 2.

5. Conclusion

We have shown that strong typing may be viewed as a purely syntactic means to restrict the class of expressions so that a nice semantic property holds. This view is consistent with practice where types are semantically (i.e. during run-time) redundant and values are really untyped.

The explicit formulation of the usefulness of strong typing makes it possible to discuss formally whether strong typing is desirable, provides a clear goal to aim at in the design of a type system, and enables a formal proof that a language, which claims to be strongly typed, satisfies that property. Thus we have a framework to discuss the type systems of [15], of ALGOL 68 and of modern languages with highly advanced type systems like LAWINE [18].

For example, [15] extends the λ-notation with a facility to pass types as a parameter. It presents no problems at all to extend our definitions, theorems and proofs to cover that extension too, see [4]. On the other hand the decision in ALGOL 68 that **struct(real** *re, im*) and **struct(real** *rho, phi*) are not equivalent seems irrelevant to maintain the representational independence of equivalence. Here, we think the ALGOL 68 designers have (mis)used the concept of strong typing in order to achieve in this particular case and in an ad-hoc way that those modes are more or less primitive. A facility to declare a type primitive, as in [15], would provide a more general solution, with no need to break the full structural equivalence of modes.

Of course, before we can make precise the above claims, further investigation is needed to extend the concepts of this paper to other language features. The introduction of cartesian product and discriminated union, and of variables and assignment, seems to be straightforward. More attention is needed for subtypes. And recursively defined types are problematic. E.g. the definitions cannot easily be adapted for the type $z = z \rightarrow z$. However, we conjecture that adaptations are possible for reducing types [1] like

$$fct = fct \times int \rightarrow int$$

which may be used to define the factorial function in the following way.

$$f : fct = \lambda g : fct, i : int\,.\ \textbf{if}\ i = 0\ \textbf{then}\ 1\ \textbf{else}\ i * g(g, i-1);$$

$$fact : int \rightarrow int = \lambda i : int\,.\ f(f, i).$$

Acknowledgement

I am grateful to Joost Engelfriet for stimulating and helpful discussions. He has also pointed out a serious error in earlier versions of this paper.

References

[1] E. Astesiano and G. Costa, Languages with reducing reflexive types, in: J.W. de Bakker and J. van Leeuwen (Eds.), Automata Languages and Programming, Lecture Notes in Computer Science, Vol. 85 (Springer, Berlin, 1980) pp. 38–50.
[2] Department of Defense (U.S.A.), Requirements for high order computer programming languages (1978).
[3] J. Donahue, On the semantics of "data type", Siam J. Comput. 8 (4) (1979) 546–560.
[4] M.M. Fokkinga, in preparation.
[5] C.A.R. Hoare, Notes on data structuring, in: O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (Eds.), Structured Programming (Academic Press, London, 1972).
[6] C.H.A. Koster, The mode-system in ALGOL 68, in: S.A. Shuman (Ed.), New Directions in Algorithmic Languages 1975 (IRIA, 78150 Le Chesnay, 1975) pp. 99–114.
[7] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (1964) 308–320.
[8] L. Meertens, Mode and meaning, in: S.A. Shuman (Ed.), New Directions in Algorithmic Languages 1975 (IRIA, 78150 Le Chesnay, 1975) pp. 125–138.
[9] R.E. Milne and C. Strachey, A Theory of Programming Language Semantics (Chapman & Hall, London, 1976).
[10] R. Milner, Fully abstract models of typed λ-calculi, Theor. Comput. Sci. 4 (1977) 1–22.
[11] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (3) (1978) 348–375.
[12] J.H. Morris, Types are not sets, in: Proc. ACM Symp. on Principles of Programming Languages, Boston, IL (1973) pp. 120–124.
[13] J.H. Morris, Towards more flexible type systems, in: B. Robinet (Ed.), Proc. Programming Symposium, Lecture Notes in Computer Science, Vol. 19 (Springer, Berlin, 1974) pp. 377–384.
[14] W.J. van der Poel, C.C. Schaap and G. van der Mey, New arithmetical operators in the theory of combinators, Indag. Math. 42 (1980) 3.
[15] J.C. Reynolds, Towards a theory of type structure, in: Proc. Programming Symposium, Lecture Notes in Computer Science Vol. 19 (Springer, Berlin, 1974) pp. 408–425.

[16] D. Scott, Data types as lattices, SIAM J. Comput. 5 (3) (1976) 522–587.
[17] J.E. Stoy, Denotational Semantics – The Scott–Strachey Approach to Programming Language Theory (MIT Press, Cambridge, MA, 1977).
[18] S.D. Swierstra, Lawine, an experiment in language and machine design, Doctoral Thesis (Twente University of Technology, The Netherlands, 1981).
[19] A. van Wijngaarden et al., Revised report on the algorithmic language ALGOL 68, Acta Informat. 5 (1975) Fasc 1–3.