stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 170/81 JULI
(DEPARTMENT OF COMPUTER SCIENCE)

J.W. DE BAKKER, J.W. KLOP & J.-J.CH. MEYER

CORRECTNESS OF PROGRAMS WITH FUNCTION PROCEDURES

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B10, 68C01, 68C05

ACM Computer Reviews Categories: 5.24, 4.22

Correctness of programs with function procedures[*)]
(extended abstract)

by

J.W. de Bakker, J.W. Klop & J.-J.Ch. Meyer[**)]

ABSTRACT

The correctness of programs with programmer-declared functions is investigated. We use the framework of the typed lambda calculus with explicit declaration of (possibly recursive) functions. Its expressions occur in the statements of a simple language with assignment, composition and conditionals. A denotational and an operational semantics for this language are provided, and their equivalence is proved. Next, a proof system for partial correctness is presented, and its soundness is shown. Completeness is then established for the case that only call-by-value is allowed. Allowing call-by-name as well, completeness is shown only for the case that the type structure is restricted, and at the cost of extending the language of the proof system. The completeness problem for the general case remains open. In the technical considerations, an important role is played by a reduction system which essentially allows us to reduce expression evaluation to systematic execution of auxiliary assignments. Termination of this reduction system is shown using Tait's computability technique. Complete proofs will appear in the full version of the paper.

KEY WORDS & PHRASES: *typed lambda calculus, recursive procedures, call-by-value, call-by-name, denotational semantics, operational semantics, Hoare's logic, soundness and completeness*

---

*) This report will be submitted for publication elsewhere.

**) Department of Computer Science, Free University, De Boelelaan 1081, 1007 MC Amsterdam.

1. INTRODUCTION

We present a study of partial correctness of programs with programmer-declared functions. Typically, if "fac" is declared as the factorial function, we want to be able to derive formulae such as $\{x=3\}$ y := fac(x)$\{y=6\}$. For this purpose, we use a functional language with an interesting structure, viz. the typed lambda calculus together with explicit declaration of (possibly recursive) functions - rather than using the fixed point combinator - and then consider a simple imperative language the expressions of which are taken from this functional language. The reader who is not familiar with the typed lambda calculus may think of function procedures as appearing in ALGOL 68, provided only finite (not recursively declared) modes are used.

Section 2 first introduces the syntax of our language(s). As to the functional language, besides constants and variables it contains *application*, two forms of *abstraction*, viz. with call-by-value and call-by-name parameters, and conditional expressions. The imperative language has assignment, composition and conditional statements. A program consists of a statement accompanied by a list of function declarations. The assignment statement constitutes our main tool in applying a formalism in the style of Hoare to an analysis of correctness of programs with function procedures. A central theme of the paper is the reduction of expression evaluation to execution of a sequence of assignment statements, thus allowing the application of the well-known partial correctness formalism for imperative languages. Some further features of our language are: function evaluation has no side-effects, the bodies of function declarations may contain global variables, and the static scope rule is applied. Section 2 also provides a denotational semantics for the language, with a few variations on the usual roles of environments and states, and applying the familiar least fixed point technique to deal with recursion.

Section 3 presents an important technical idea. A system of *simplification rules* is given for the statements of our language allowing the reduction of each statement to an equivalent *simple* one. These rules embody the above-mentioned imperative treatment of expression evaluation, and play a crucial role both in the definition of the operational semantics to be given in Section 4, and in the proof systems to be studied in Sections 5 to 7. The proof that the reduction always terminates is non-trivial. Details are given in the Appendix; the proof relies on the introduction of a *norm* for each expression. The existence of this norm is proved using an auxiliary reduction system. Reduction in this auxiliary system always terminates as is shown using the "computability" technique of Tait [22]. In Section 4 we define an operational semantics for our language and prove its equivalence with the denotational one.

In Section 5 the notion of partial correctness formula is introduced, and a sound proof system for partial correctness is proposed. The techniques used in the soundness proof rely partly on the equivalence result of Section 4, partly follow the lines of De Bakker [4]. In Section 6 we show that a slight modification of the proof system is complete for a language with only call-by-value abstraction. This is shown by appropriate use of the technique of Gorelick [11], described also e.g. in Apt [1] and De Bakker [4]. Section 7 discusses completeness when call-by-value and call-by-name are combined, but only for the case that all arguments of functions are of ground type (no functions with functions as arguments). We present a complete proof system for this case, albeit at the cost of extending the language of the proof system with an auxiliary type of assignment, allowing the undefined constant in assertions, and adding to the proof system a number of proof rules exploiting the auxiliary assignment. The completeness problem for the general case (functions with functions as arguments) remains open. In the Appendix we give some details on the proof of termination of the simplification system of Section 3.

Partial correctness of programs with function procedures has not yet been investigated extensively in the literature. Clint & Hoare [8] (see also Ashcroft, Clint & Hoare [2], O'Donnell [19]) propose a rule which involves the appearance of calls of programmer-declared functions within assertions. The proof system we shall propose avoids this. A general reference for the (typed) lambda calculus is Barendregt [5]. The semantics of the typed lambda calculus has been thoroughly investigated e.g. by Plotkin [20] and - extended with nondeterminacy, by Hennessy & Ashcroft [12,13]. However, correctness issues in our sense are not addressed in these papers. (LCF [10] *is* a logical system for function procedures, but not the one of partial correctness.) The operational semantics of Section 4 follows the general pattern as proposed by Cook [9] and further analyzed by De Bruin [6]. The partial correctness formalism was introduced by Hoare [13]; many details on further developments can be found in e.g. Apt [1] or De Bakker [4]. Completeness is always taken in the sense of Cook's *relative* completeness [9]. Related work on (in)completeness of partial correctness for procedures is described in Clarke [7]; a survey paper on this topic is Langmaack & Olderog [17].

Acknowledgements. We are grateful for a number of helpful discussions with K.R. Apt, H.P. Barendregt, A. de Bruin, E.M. Clarke, H. Langmaack and E.R. Olderog. Arie de Bruin has in particular clarified some problems we had with the definition of the denotational semantics, and Ed Clarke did the same for the (in)completeness problem.

2. SYNTAX AND DENOTATIONAL SEMANTICS

Notation. For any set M, the phrase "let $(m\in)M$ be such that ..." defines M by ..., and simultaneously introduces m as typical element of M.

We first present the syntax of our language. It uses a typed lambda calculus with programmer-declared functions allowing (explicit) recursion, embedded into a simple imperative language.

The set $(\tau\in)$ *Type* is defined by $\tau ::= \omega \mid (\tau_1 \to \tau_2)$. A type τ is either *ground* $(\tau = \omega)$, *functional* ($\tau \neq \omega$, this abbreviates that $\tau = \tau_1 \to \tau_2$ for some types τ_1, τ_2), or arbitrary. Type $(\omega \to (\omega \to \ldots \to (\omega \to \omega)\ldots))$ is usually abbreviated to $\omega^n \to \omega$, $n \geq 0$.

The set (c∈) *Cons* is that of the *constants*, which are always of the type $\omega^n \to \omega$, $n \geq 0$. We use the letters x, y, z, u for variables of ground type, f, g for variables of functional type, and v, w for variables of arbitrary type. For later use, we assume the respective sets of variables to be well-ordered. In the intended meaning, (function) constants are given initially (as part of some given signature, if one prefers) and assigned values - by some interpretation - in a set $V_\omega^n \to V_\omega$, V_ω the set of ground values. For example, taking V_ω as the set of integers, "+" might be the interpretation of a constant of type $\omega^2 \to \omega$. Function variables are to be programmer-declared ("fac" above is an example). Note that, contrary to the situation for constants, their arguments may themselves be functions (type $((\omega\to\omega) \to \omega)$ is an example).

The set of expressions is defined as follows: First we give the syntax for the untyped expressions (s,t∈) *Uexp*. After that, we present the typing rules which determine the subset *Exp* consisting of all expressions which can be typed according to these rules. From that moment on, s,t always stand for typed expressions.

$$s ::= c \mid v \mid s_1(s_2) \mid \langle \underline{\text{val}}\ x{:}s\rangle \mid \langle \underline{\text{name}}\ v{:}s\rangle \mid \underline{\text{if}}\ b\ \underline{\text{then}}\ s_1\ \underline{\text{else}}\ s_2\ \underline{\text{fi}}$$

(We take this syntax in the sense that wherever an arbitrary variable v may appear, also x,... or f,... may appear). The following formulae suggest the typing rules (s^τ is to be read here as: s is of type τ): (i) c^τ, where $\tau = \omega^n \to \omega$, $n \geq 0$ (ii) x^ω, v^τ for any τ), f^τ for $\tau \neq \omega$ (iii) $(s_1^{\tau_1\to\tau_2}(s_2^{\tau_1}))^{\tau_2}$ (iv) $\langle \underline{\text{val}}\ x^\omega{:}\ s^\tau\rangle^{\omega\to\tau}$ (v) $\langle \underline{\text{name}}\ v^{\tau_1}{:}\ s^{\tau_2}\rangle^{\tau_1\to\tau_2}$ (vi) $(\underline{\text{if}}\ b\ \underline{\text{then}}\ s_1^\tau\ \underline{\text{else}}\ s_2^\tau\ \underline{\text{fi}})^\tau$.

Examples

1. Expressions which cannot be typed:
   $x(y)$, $v(v)$, $c(x)(f)$, $\langle \underline{\text{val}}\ x{:}s\rangle(f)$
2. Expressions which can be typed:
   $f(y)$, $c(x)(y)$, $\langle \underline{\text{val}}\ x{:}c\rangle(y)$, $\langle \underline{\text{val}}\ x{:}\ \langle \underline{\text{name}}\ f{:}\ f(x)\rangle\rangle(c)(g)$, $\underline{\text{if}}\ b\ \underline{\text{then}}\ c_0\ \underline{\text{else}}\ c_1(x)(f(c_2(x)))\ \underline{\text{fi}}$

For simplicity's sake, we only treat call-by-value parameters of *ground* type in our language (whereas call-by-name parameters are arbitrary). When confusion is unlikely, we simply use s instead of s^τ.

As further syntactic categories we introduce (b∈) *Bexp* (boolean expressions), (S∈) *Stat* (statements), (D∈) *Decl* (declarations), (P∈) *Prog* (programs), and (e∈) *Sexp* (simple expressions) as follows:

$$b ::= \underline{\text{true}} \mid s_1^\omega = s_2^\omega \mid \neg b \mid b_1 \supset b_2$$

$$S ::= x{:=}s \mid S_1;S_2 \mid \underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}$$

$$D ::= f_1^{\tau_1} \Leftarrow t_1^{\tau_1}, \ldots, f_n^{\tau_n} \Leftarrow t_n^{\tau_n}, \quad n \geq 0$$

$$P ::= \langle D{:}S\rangle$$

$$e ::= x \mid c(e_1)\ldots(e_n), \quad n \geq 0$$

Some further terminology and explanation about syntax is provided in the

Remarks

1. "$\equiv$" will denote syntactic identity
2. $<D:S>$ is usually written as $<D|S>$
3. An example of a program (for suitably chosen constants) is
   $<f \Leftarrow <$<u>val</u> x: <u>if</u> x=0 <u>then</u> y <u>else</u> x*f(x-1) <u>fi</u> | y:=1; z:=f(2)>
4. A variable v^τ is *bound* in a program either by abstraction or (for $\tau \neq \omega$) by appearing on the left-hand side of a function declaration. Var_τ is the set of all variables of type τ. $\mathit{var}_\omega(s)$ denotes the set of all free ground variables of s. $x \in \mathit{var}_\omega(s)$ will sometimes be abbreviated to $x \in s$. Similar notations such as $\mathit{var}_\omega(D,S)$, $\mathit{var}_\tau(s)$, $f \in s$, etc. should be clear. A program $P \equiv <D|S>$ is called *closed* whenever $\mathit{var}_\tau(P) = \emptyset$ for $\tau \neq \omega$.
5. In a closed program $<D|S>$, the only free variables are of ground type. In programming terminology, these are the *globals* of the program. They appear either in the body of function declarations (in t_j, where $D \equiv <f_i \Leftarrow t_i>_i$) or in S (e.g., in x := <u>&lt;val</u> u: c(u)(y)>(z), the globals are x, y, z).
6. *Substitution* of t for v in s is denoted by s[t/v]. The usual precautions to avoid clashes between free and bound variables apply.
7. An n-tuple $(s_1)\ldots(s_n)$, $n \geq 0$, is often abbreviated to $\vec{s}_{1:n}$ or to $\vec{s}$. $\vec{s}_{2:n}$ denotes $(s_2)\ldots(s_n)$; also, a notation such as $(y:=s)^{\rightarrow}$ is short for $y_1:=s_1;\ldots;y_n:=s_n$, $n \geq 0$.
8. *Simple* expressions e (are always of ground type and) have no function calls or abstraction; they are therefore essentially simpler than arbitrary expressions, and play a certain "atomic" rôle in the subsequent considerations.

In the *semantics* we introduce domains $(\phi^\tau \in)\ V_\tau$, for each τ, as follows: let V_0 be some arbitrary set, and let $(\alpha\in)V_\omega = V_0 \cup \{\perp_\omega\}$ be the *flat* cpo of ground values over V_0 (i.e., $\alpha_1 \sqsubseteq \alpha_2$ iff $\alpha_1 = \perp_\omega$ or $\alpha_1 = \alpha_2$). Let $V_{\tau_1\to\tau_2} = [V_{\tau_1} \to V_{\tau_2}]$, i.e., all *continuous* functions $V_{\tau_1} \to V_{\tau_2}$, and let $\perp_\tau$ denote the least element of V_τ (i.e., for $\tau = \tau_1 \to \tau_2$, $\perp_\tau = \lambda\phi^{\tau_1}\cdot\perp_{\tau_2}$). Let $(\beta\in)W = \{ff,tt\} \cup \{\perp_W\}$ be the flat cpo of *truth-values*. Let $\Sigma_0 = \mathit{Var}_\omega \to V_0$, and let $(\sigma\in)\Sigma = \Sigma_0 \cup \{\perp_\Sigma\}$ be the flat cpo of *states*. Let $(\eta^\tau\in)N_\tau = \Sigma \to_s V_\tau$, $\to_s$ denoting *strict* functions, be the cpo ordered by $\eta_1^\tau \sqsubseteq \eta_2^\tau$ iff $\eta_1^\tau(\sigma) \sqsubseteq \eta_2^\tau(\sigma)$ for all σ. Let $(\eta\in)N = \cup_\tau N_\tau$, and let $\mathit{Var} = \cup_\tau \mathit{Var}_\tau$. *Environments* ε are functions: $\mathit{Var} \to N$ which are used primarily to assign meanings either to the variables appearing as parameters in abstraction, or to declared function variables. (Note that $\varepsilon(f) \in N$ in general depends on the state since the meaning of f may be changed by assignment to some global variable, such as "y:=1" in the example of remark 3 above.) For technical reasons, it is convenient to address *all* ground variables through ε. This is achieved by the following definitions:

1. ε is called *normal in* x iff $\varepsilon(x) = \lambda\sigma\cdot\sigma(x)$ (i.e., $\varepsilon(x)(\sigma) = \sigma(x)$: normally, the value of a variable is obtained by applying the state to it)

2. ε is said to *store* x iff $\varepsilon(x) = \lambda\sigma\cdot\alpha$, for some α (i.e., $\varepsilon(x)(\sigma) = \alpha$: α is the value - which may be $\bot_\omega$ - stored for the formal parameter x (see Def. 2.1) and is independent of the state)
3. The set *Env* of environments is defined as
   $Env = \{\varepsilon \in Var \to N \mid \varepsilon(Var_\tau) \subseteq N_\tau$, and $\forall x[\varepsilon$ is normal in x or ε stores x$]\}$.
4. ε is called *normal* iff $\varepsilon(x) = \lambda\sigma\cdot\sigma(x)$ for *all* x.

We note that, for ε normal in x, $\varepsilon(x)(\sigma)(= \sigma(x)) \neq \bot_\omega$ is always satisfied for $\sigma \neq \bot_\Sigma$, whereas, if ε stores x, we may have that $\varepsilon(x)(\sigma) = \bot_\omega$, for $\sigma \neq \bot_\Sigma$.
Two further pieces of notation are needed:

1. We shall use $\tilde{\lambda}\alpha\cdot\phi^{\omega\to\tau}(\alpha)$ as notation for the *strict* function defined by $\lambda\alpha\cdot$ <u>if</u> $\alpha = \bot_\omega$ <u>then</u> $\bot_\tau$ <u>else</u> $\phi^{\omega\to\tau}(\alpha)$ <u>fi</u>
2. For $\alpha \neq \bot_\omega$ and $\sigma \neq \bot_\Sigma$, $\sigma\{\alpha/x\}$ denotes the state such that
   $$\sigma\{\alpha/x\}(y) = \begin{cases} \alpha, & \text{if } x \equiv y \\ \sigma(y), & \text{if } x \not\equiv y \end{cases}$$
   . Similarly for $\varepsilon\{\eta/v\}$ etc.

In the denotational semantics we first fix an interpretation J for all c:
J: *Cons* $\to (V_\omega \to_s (V_\omega \to_s \dots \to_s (V_\omega \to_s V_\omega)\dots))$. Note that the meaning of a constant is always a *strict* function. As valuation functions for the various syntactic classes we introduce

V: *Exp* $\to$ (*Env* $\to (\Sigma \to_s V))$
W: *Bexp* $\to$ (*Env* $\to (\Sigma \to_s W))$
M: *Stat* $\to$ (*Env* $\to (\Sigma \to_s \Sigma))$
N: *Prog* $\to$ (*Env* $\to (\Sigma \to_s \Sigma))$

(E for *Sexp* is given later). They are defined in

DEFINITION 2.1. (denotational semantics)

a. $V(s^\tau)(\varepsilon)(\bot_\Sigma) = \bot_\tau$, and for $\sigma \neq \bot_\Sigma$,
   $V(v)(\varepsilon)(\sigma) = \varepsilon(v)(\sigma)$
   $V(c)(\varepsilon)(\sigma) = J(c)$
   $V(s_1(s_2))(\varepsilon)(\sigma) = V(s_1)(\varepsilon)(\sigma)(V(s_2)(\varepsilon)(\sigma))$
   $V($<<u>val</u> x:s>$)(\varepsilon)(\sigma) = \tilde{\lambda}\alpha\cdot V(s)(\varepsilon\{\tilde{\lambda}\sigma\cdot\alpha/x\})(\sigma)$
   $V($<<u>name</u> v:s>$)(\varepsilon)(\sigma) = \lambda\phi\cdot V(s)(\varepsilon\{\tilde{\lambda}\sigma\cdot\phi/v\})(\sigma)$
   $V($<u>if</u> b <u>then</u> s_1 <u>else</u> s_2 <u>fi</u>$)(\varepsilon)(\sigma) =$
   <u>if</u> $W(b)(\varepsilon)(\sigma)$ <u>then</u> $V(s_1)(\varepsilon)(\sigma)$ <u>else</u> $V(s_2)(\varepsilon)(\sigma)$ <u>fi</u>

b. $W(b)(\varepsilon)(\bot_\Sigma) = \bot_W$, and, for $\sigma \neq \bot_\Sigma$,
   $$W(s_1{=}s_2)(\varepsilon)(\sigma) = \begin{cases} \bot_W, & \text{if } V(s_1)(\varepsilon)(\sigma) = \bot_\omega \text{ or } V(s_2)(\varepsilon)(\sigma) = \bot_\omega \\ V(s_1)(\varepsilon)(\sigma) = V(s_2)(\varepsilon)(\sigma), & \text{otherwise} \end{cases}$$
   (other clauses are simple and omitted)

c. $M(x{:=}s)(\varepsilon)(\sigma) =$ <u>if</u> $V(s)(\varepsilon)(\sigma) = \bot_\omega$ <u>then</u> $\bot_\Sigma$ <u>else</u> $\sigma\{V(s)(\varepsilon)(\sigma)/x\}$ <u>fi</u>
   $M(S_1;S_2)(\varepsilon)(\sigma) = M(S_2)(\varepsilon)(M(S_1)(\varepsilon)(\sigma))$

$\mathcal{M}(\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}})(\varepsilon)(\sigma) =$
$\underline{\text{if}}\ \mathcal{W}(b)(\varepsilon)(\sigma)\ \underline{\text{then}}\ \mathcal{M}(S_1)(\varepsilon)(\sigma)\ \underline{\text{else}}\ \mathcal{M}(S_2)(\varepsilon)(\sigma)\ \underline{\text{fi}}$

d. $\mathcal{N}(\langle\langle f_i \Leftarrow t_i\rangle_i \mid S\rangle)(\varepsilon)(\sigma) = \mathcal{M}(S)(\varepsilon\{\eta_i/f_i\}_i)(\sigma)$, where $\langle\eta_1,\dots,\eta_n\rangle = \mu[H_1,\dots,H_n]$, and $H_j = \lambda\eta'_1 \cdot \ldots \cdot \lambda\eta'_n.\ \mathcal{V}(t_j)(\varepsilon\{\eta'_i/f_i\}_i)$, $j = 1,\dots,n$.

Remarks.

1. Note that the assignment is strict in the value of its right-hand side. Call-by-value abstraction is strict as well; this observation forms the starting point for the simplification of the next section.
2. It can be shown that $\mathcal{V}(s^\tau)(\varepsilon)(\sigma) \in V_\tau$.
3. $\mu[H_1,\dots,H_n]$ in clause d denotes the simultaneous least fixed point of the operators $H_1,\dots,H_n$; its existence follows from the continuity of each of the H_j, $j = 1,\dots,n$.

3. SIMPLIFICATION

We first observe that an analysis of the structure of the expressions s^ω yields that each assignment $x := s^\omega$ is one of the following forms.

1. $x := e$
2. $x := c\vec{s}_{1:n}$ $(n\geq 1)$, not all s_i simple
3. $x := f\vec{s}_{1:n}$ $(n\geq 1)$,
4. $x := \langle\underline{\text{val}}\ x'{:}s_0\rangle\vec{s}_{1:n}$ $(n\geq 1)$,
5. $x := \langle\underline{\text{name}}\ v{:}s_0\rangle\vec{s}_{1:n}$ $(n\geq 1)$,
6. $x := \underline{\text{if}}\ b\ \underline{\text{then}}\ s'\ \underline{\text{else}}\ s''\ \underline{\text{fi}}\ \vec{s}_{1:n}$ $(n\geq 0)$.

Moreover, we recall that, in 2, all the s_i are of ground type, and in 4 s_1 is of ground type.

We next define the notion of a *simple* statement T by:

$$T ::= x{:=}e \mid x{:=}f\vec{s} \mid T_1;T_2 \mid \underline{\text{if}}\ e_1{=}e_2\ \underline{\text{then}}\ T_1\ \underline{\text{else}}\ T_2\ \underline{\text{fi}}$$

(Essentially, in a simple statement we have eliminated abstraction outside the arguments of function variables.) We now present a system of reduction rules allowing us to reduce each statement to an equivalent (but for the values of certain auxiliary variables) simple statement. Let us call the reduction system $\mathcal{RS}_1$. It has the rules

1. $x{:=}c\vec{s} \rightsquigarrow \overrightarrow{(y{:=}s)};\ x{:=}c\vec{y}$, not all s_i simple
2. $x{:=}\langle\underline{\text{val}}\ x'{:}s_0\rangle\vec{s} \rightsquigarrow y{:=}s;\ x{:=}s_0[y/x']\vec{s}_{2:n}$
3. $x{:=}\langle\underline{\text{name}}\ v{:}s_0\rangle\vec{s} \rightsquigarrow x{:=}s_0[s_1/v]\vec{s}_{2:n}$
4. $x{:=}\underline{\text{if}}\ b\ \underline{\text{then}}\ s'\ \underline{\text{else}}\ s''\ \underline{\text{fi}}\ \vec{s} \rightsquigarrow \underline{\text{if}}\ b\ \underline{\text{then}}\ x{:=}s'\vec{s}\ \underline{\text{else}}\ x{:=}s''\vec{s}\ \underline{\text{fi}}$
5. $\underline{\text{if}}\ \underline{\text{true}}\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}} \rightsquigarrow S_1$
6. $\underline{\text{if}}\ \neg b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}} \rightsquigarrow \underline{\text{if}}\ b\ \underline{\text{then}}\ S_2\ \underline{\text{else}}\ S_1\ \underline{\text{fi}}$
7. $\underline{\text{if}}\ b_1 \supset b_2\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}} \rightsquigarrow$
   $\underline{\text{if}}\ b_1\ \underline{\text{then}}\ \underline{\text{if}}\ b_2\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\ \underline{\text{else}}\ S_1\ \underline{\text{fi}}$

8. $\underline{\text{if}}\ s_1=s_2\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}} \rightsquigarrow$
$y_1:=s_1;\ y_2:=s_2;\ \underline{\text{if}}\ y_1=y_2\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}$, not all s_i simple

In rule 1 all (ground) arguments of a function constant are evaluated and stored in auxiliary variables. $x:=c\vec{y}$ is an assignment with a *simple* right-hand side, and the s_i are to be subjected to further reduction. Rule 2 expresses call-by-value through assignment. Rule 3 deals with call-by-name through substitution (it is the rule of β- conversion of the lambda calculus). Rules 4 to 8 are self-explanatory. In subsequent applications of RS_1, we shall employ it in the context of a declaration D. We shall then impose upon the y_i the constraints that, in each of the rules 1, 2, and 8 we have:

(i) all y_i are different and do not appear free on the left-hand side of the rule.
(ii) none of the y_i appears free in D.

THEOREM 3.1.

a. The reduction system RS_1 always terminates transforming each statement S to some simple T.
b. In case restrictions (i), (ii) on the choice of the y_i are imposed, T is equivalent to S, but for the values of the auxiliary variables.

Proof. An outline of the proof of part a can be found in the Appendix. The problem is nontrivial since induction on the syntactic complexity of s (in x:=s) does not work directly (because of rule 3), and a suitable means for bringing the type structure into the picture has to be found. The proof of part b is implicit in the considerations of Section 4. □

4. OPERATIONAL SEMANTICS AND THE EQUIVALENCE THEOREM

In the operational semantics we start from the evaluation of simple expressions through the valuation function E: $Sexp \to (\Sigma_0 \to V_0)$ defined by $E(x)(\sigma) = \sigma(x)$, $E(c(e_1)\dots(e_n))(\sigma) = J(c)(E(e_1)(\sigma))\dots(E(e_n)(\sigma))$. Next, following the approach of Cook [9], we define a (partial) function C: $Prog \to_{part} (\Sigma \to \Sigma^\infty)$, where $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, the set of all finite or infinite sequences of states. In the definition we use "^" for concatenation of sequences (on the right-hand side of an infinite sequence it has no effect) and κ: $\Sigma^\infty \to \Sigma$ for the function yielding the last element of a sequence if it exists and $\bot_\Sigma$ otherwise. For ρ a sequence in Σ^∞, if $\rho = <\sigma_1,\dots,\sigma_n> \in \Sigma$, then $\rho\{\alpha/y\}$ denotes $<\sigma_1,\dots,\sigma_{n-1}, \sigma_n\{\alpha/y\}>$, and if $\rho \in \Sigma^\omega$ then $\rho\{\alpha/y\} = \rho$.

We now give the rules for $C(<D|S>)$. For brevity, we rather write $C_D(S)$:

DEFINITION 4.1. (computation sequences)

1. $C_D(S)(\bot_\Sigma) = <\bot_\Sigma>$, and, for $\sigma \neq \bot_\Sigma$,

2a. $C_D(x:=e)(\sigma) = <\sigma\{E(e)(\sigma)/x\}>$

b. $C_D(x:=f\vec{s})(\sigma) = <\sigma>$^$C_D(x:=t\vec{s})(\sigma)$, if $f \Leftarrow t$ occurs in D (otherwise $C_D(x:=f\vec{s})(\sigma)$ is undefined)

c. $C_D(T_1;T_2)(\sigma) = \langle\sigma\rangle\verb|^|C_D(T_1)(\sigma)\verb|^|C_D(T_2)(\kappa(C_D(T_1)(\sigma)))$

d. $C_D(\underline{\text{if}}\ e_1=e_2\ \underline{\text{then}}\ T_1\ \underline{\text{else}}\ T_2\ \underline{\text{fi}})(\sigma) = \begin{cases}\langle\sigma\rangle\verb|^|C_D(T_1)(\sigma), & \text{if } E(e_1)(\sigma) = E(e_2)(\sigma)\\ \langle\sigma\rangle\verb|^|C_D(T_2)(\sigma), & \text{if } E(e_1)(\sigma) \neq E(e_2)(\sigma)\end{cases}$

3. $C_D(S)(\sigma) = \langle\sigma\rangle\verb|^|C_D(S')(\sigma)\{\sigma(y_i)/y_i\}_i$, for each rule $S \rightsquigarrow S'$ of the system RS_1, where $y_1,\ldots,y_n$, $n \geq 0$, are the auxiliary variables introduced in that rule (for uniqueness, we assume that the first y_i satisfying restrictions (i), (ii) are chosen).

Remark. In clause 3, the auxiliary variables - after having served their purpose as temporary storage - are reset to their original values by the modification $\{\sigma(y_i)/y_i\}_{i=1}^n$.

LEMMA 4.2. The system of equations for C_D given in Definition 4.1 has a unique solution.

Proof. The proof combines the techniques as described in De Bruin [6] with some ideas from the termination proof for RS_1. □

DEFINITION 4.3. The operational semantics O: *Prog* $\to_{\text{part}} (\Sigma \to_s \Sigma)$ is defined by $O(\langle D|S\rangle) = \kappa\circ C_D(S)$.

Again, we shall often write $O_D(S)$ for $O(\langle D|S\rangle)$. The denotational and operational semantics for closed programs coincide. In order to prove this, some auxiliary notions and results are necessary, some of which also play a part in subsequent sections. (For some background concerning the notion "does not use" see De Bakker [4].)

LEMMA 4.4. (first substitution lemma)

a. $V(s[t/f])(\varepsilon) = V(s)(\varepsilon\{V(t)(\varepsilon)/f\})$

b. $V(s[t/v])(\varepsilon) = \lambda\sigma\cdot V(s)(\varepsilon\{\lambda\tilde{\sigma}\cdot V(t)(\varepsilon)(\sigma)/v\})(\sigma)$

Remark. Note that using v instead of f in part a would (for v of ground type) not be well-formed because of the definition of *Env*.

DEFINITION 4.5.

a. A function η *does not use* x iff for all σ, $\alpha \neq \bot_\omega$, $\eta(\sigma\{\alpha/x\}) = \eta(\sigma)$

b. ε does not use x iff $\varepsilon(f)$ does not use x for all f.

LEMMA 4.6. Let $D = \langle f_i \Leftarrow t_i\rangle_i$, and let η_i be as in Def. 2.1.

a. If $x \notin D$, s and ε does not use x then $V(s)(\varepsilon\{\eta_i/f_i\}_i)$ does not use x.

b. Let $\langle D|s\rangle$ be closed. If $x \notin D$, s then $V(s)(\varepsilon\{\eta_i/f_i\}_i)$ does not use x.

Remark. By way of explanation of clause b we observe that a function which is the meaning of an expression *uses* a variable only if it occurs in the expression directly or indirectly (i.e., as a global of a function procedure called in the expression).

LEMMA 4.7. (second substitution lemma)

a. $V(s[y/x])(\varepsilon)(\sigma\{\alpha/y\}) = V(s)(\varepsilon\{\lambda\tilde{\sigma}\cdot\alpha/x\})(\sigma)$, provided that $y \notin var_\omega(s)\backslash\{x\}$, ε is *normal* in x, y and ε does not use y.

b. $V(s[t/x])(\varepsilon)(\sigma) = V(s)(\varepsilon)(\sigma\{V(t)(\varepsilon)(\sigma)/x\})$ provided that ε is normal in x and ε does not use x.

We now exhibit a series of facts leading to the equivalence theorem. We always assume that $D \equiv \langle f_i \Leftarrow t_i\rangle_i$.

LEMMA 4.8. For $\langle D|S\rangle$ closed and ε normal, $O_D(S) \sqsubseteq N_D(S)(\varepsilon)$.

Proof. Induction on the length of the computation sequence used (by $C_D(S)$) to determine $O_D(S)$.

The reverse inclusion takes more effort and is proved in a number of steps. Assume again that $\langle D|S\rangle$ is closed and ε normal. We introduce some auxiliary syntax. For each τ, let Ω^τ be the nowhere defined expression (i.e., $V(\Omega^\tau)(\varepsilon)(\sigma) = \bot_\tau$, and $C_D(x:=\Omega^{\tau\to}s)(\sigma) = \langle\bot_\Sigma\rangle$). With respect to D we define $S^{(j)}$ and $S^{[j]}$, $j = 0,1,\ldots$: *Notation*. $S^{(0)} \equiv S$, $S^{(j+1)} \equiv S^{(j)}[t_i/f_i]_i$, $S^{[j]} \equiv S^{(j)}[\Omega/f_i]_i$, $j = 0,1,\ldots$. Let $\varepsilon_k = \varepsilon\{\eta_i^k/f_i\}_i$, $k = 0,1,\ldots$, where $\eta_i^0 = \bot$, $\eta_j^{k+1} = V(t_j)(\varepsilon\{\eta_i^k/f_i\}_i)$, $j = 1,\ldots,n$. For η_i as in Def. 2.1, by continuity we have that $\eta_i = \bigsqcup_k \eta_i^k$, $i = 1,\ldots,n$.

Step

1.	$N_D(S)(\varepsilon) = M(S)(\varepsilon\{\eta_i/f_i\}_i)$	(def. 2.1)
2.	$M(S)(\varepsilon\{\eta_i/f_i\}_i) = \bigsqcup_k M(S)(\varepsilon_k)$	(continuity)
3.	$M(S)(\varepsilon_k) = M(S^{[k]})(\varepsilon)$	(lemma 4.4)

4. $M(S^{[k]})(\varepsilon) \sqsubseteq O_D(S^{[k]})$

   This uses induction on the norm of $S^{[k]}$ as introduced in the Appendix, and the requirement that ε be normal. Note that, since $\langle D|S\rangle$ is closed, $S^{[k]}$ does not contain free function variables (for $\tau \neq \omega$, $var_\tau(S) \subseteq \{f_1,\ldots,f_n\}$; hence, $var_\tau(S^{[k]}) = \emptyset$).

5. If $O_D(S^{[k]})(\sigma) = \sigma' \neq \bot_\Sigma$, then $O_D(S^{(k)})(\sigma) = \sigma'$.

   This is a familiar argument, cf. the "genericity" result of Barendregt [5]. E.g., if $x:=s[\Omega/f] \equiv x:=\ldots\Omega\ldots$, for input σ yields output $\sigma' \neq \bot_\Sigma$, then Ω is not encountered during execution of s (it never appears in the head position), and execution of $x:=\ldots\Omega\ldots$may just as well be replaced by execution of $x:=s \equiv x:=\ldots f\ldots$

6. $O_D(S^{(k)}) = O_D(S)$.

   This is the fixed point property of (recursively declared) function procedures. A special case is that $O_D(x:=s[t/f]) = O_D\ (x:=s)$, for $f \Leftarrow t$ in D.

Combining steps 1 to 6 we obtain that $N_D(S)(\varepsilon) \sqsubseteq O_D(S)$; together with Lemma 4.8 this yields

THEOREM 4.9. For $\langle D|S\rangle$ closed and ε normal, $N_D(S)(\varepsilon) = O_D(S)$.

5. CORRECTNESS AND A SOUND PROOF SYSTEM

We are interested in proving facts such as

$$\models \langle f \Leftarrow \langle \underline{\text{val}}\ x\text{: } \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x*f(x-1)\ \underline{\text{fi}} \mid \{x=3\}y:=f(x)\{y=6\}\rangle.$$

Note that in this example a postcondition $y = f(3)$, though trivially true, would not be particularly helpful. In order to avoid this phenomenon (assertions with unevaluated function calls), we restrict ourselves to assertions with only *simple* expressions (no function calls, and no abstraction either). The class $(p,q,r\epsilon)$ Assn is defined by

$$p ::= \underline{\text{true}}\ |e_1=e_2|\neg p|p_1\supset p_2|\exists x[p]$$

The valuation $\mathcal{T}$: $\mathit{Assn} \rightarrow (\Sigma\rightarrow_s\{tt,ff\})$ is defined by $\mathcal{T}(p)(\bot_\Sigma) = ff$, and, for $\sigma \neq \bot_\Sigma$, $\mathcal{T}(e_1=e_2)(\sigma) = (\mathcal{E}(e_1)(\sigma) = \mathcal{E}(e_2)(\sigma)),\ldots$ (the other clauses are obvious). A *correctness formula* is a construct of the form $\langle D|F_1\Rightarrow F_2\rangle$, where F_1, F_2 are conjunctions of assertions and triples $\{p\}S\{q\}$. An example is the rule of consequence $\langle D|(p\supset p_1) \wedge \{p_1\}S\{q_1\} \wedge \wedge(q_1\supset q) \Rightarrow \{p\}S\{q\}\rangle$. $\langle D|F\rangle$ abbreviates $\langle D|\underline{\text{true}} \Rightarrow F\rangle$. Correctness formulae contain D to provide declarations for the (functions called in the) S appearing in them. The "⇒" formalism leads to a system in the style of Gentzen's "sequent calculus" (rather than Gentzen's natural deduction) to deal with recursion. Further explanation of this can be found e.g. in Apt [1] or De Bakker [4]. For the definition of validity we first provide a valuation $\mathcal{F}$, assigning meaning to $\{p\}S\{q\}$ in the usual manner:

$$\mathcal{F}(\{p\}S\{q\})(\varepsilon)(\sigma) = \forall\sigma'[\mathcal{T}(p)(\sigma) \wedge \sigma' = \mathcal{M}(S)(\varepsilon)(\sigma) \wedge \sigma' \neq \bot_\Sigma \Rightarrow \mathcal{T}(q)(\sigma')].$$

In case we want to stress that $\mathcal{F}$ depends on $\mathcal{J}$ (the interpretation of the constants), we write $\mathcal{F}_\mathcal{J}$. We then put $\models_\mathcal{J} \langle D|F_1\Rightarrow F_2\rangle$ iff for all normal ε such that ε uses no (ground) variables not in D.

$$\forall\sigma \neq \bot[\mathcal{F}_\mathcal{J}(F_1)(\varepsilon\{\eta_i/f_i\}_i)(\sigma)] \Rightarrow \forall\sigma \neq \bot[\mathcal{F}_\mathcal{J}(F_2)(\varepsilon\{\eta_i/f_i\}_i)(\sigma)],$$

where the η_i are as in Def. 2.1.
The restrictions on ε firstly imply that all free ground variables of D, F_1, F_2 are treated as normal variables ($\varepsilon(x)(\sigma) = \sigma(x)$, for all x); moreover, for all f, $\varepsilon(f)$ uses only variables in D. For f declared in D, this is to be expected; for f undeclared (a situation stemming from the proof rule for recursion) it has to be postulated for reasons explained in Chapter 5 of De Bakker [4].

Usually, $\mathcal{J}$ is understood, and we simply write $\models$ instead of $\models_\mathcal{J}$. For simplicity's sake, in the remainder of the paper we always assume $D \equiv f \Leftarrow t$ to consist of the declaration of only one function procedure.

The proof system for partial correctness has three groups of rules. Group I has the obvious rules for "⇒" such as, e.g., transitivity of "⇒" or the fact that $\langle D|F\Rightarrow F_1\rangle$ and $\langle D|F\Rightarrow F_2\rangle$ imply that $\langle D|F\Rightarrow F_1\wedge F_2\rangle$. Group II is the central one. It has two subgroups, one providing a rule for each simple statement, one based on the simplification rules:

II a.1. $\langle D|\{p[e/x]\}x:=e\{p\}\rangle$ (assignment)

2. $\dfrac{\langle D \mid \{p\}x:=g\vec{s}\{q\} \Rightarrow \{p\}x:=t[g/f]\vec{s}\{q\}\rangle}{\langle D \mid \{p\}x:=f\vec{s}\{q\}\rangle}$ with $g \notin D, \vec{s}$. (recursion)

3. $\langle D \mid \{p\}S_1\{q\} \wedge \{q\}S_2\{r\} \Rightarrow \{p\}S_1;S_2\{r\}\rangle$ (composition)

4. $\langle D \mid \{p\wedge(e_1=e_2)\}S_1\{q\} \wedge \{p\wedge(e_1\neq e_2)\}S_2\{q\}$
$\Rightarrow \{p\}$ <u>if</u> $e_1=e_2$ <u>then</u> S_1 <u>else</u> S_2 <u>fi</u> $\{q\}\rangle$ (conditionals)

b. $\langle D \mid \{p\}S'\{q\} \Rightarrow \{p\}S\{q\}\rangle$ (simplification)

where $S \rightsquigarrow S'$ is one of the rules of the system RS_1, and the choice of the auxiliary $\vec{y}$ is further restricted by (iii) none of the $\vec{y}$ occurs free in p or q.

III In the third group, we find a number of auxiliary rules. Besides the already mentioned rule of consequence, it consists of

$\langle D \mid \{p\}S\{q_1\} \wedge \{p\}S\{q_2\} \Rightarrow \{p\}S\{q_1;q_2\}\rangle$ (conjunction)

$\langle D \mid \{p\}x:=s\{p\}\rangle$, provided that $x \notin p$ (invariance)

$\langle D \mid \{p\}x:=s\{q\} \Rightarrow \{p[z/y]\}x:=s\{q\}\rangle$ (substitution, I)
provided that $y \equiv z$ or $y \notin D,s,q$

$\langle D \mid \{p\}x:=s\{q\} \Rightarrow \{p[y/x]\}y:=s[y/x]\{q[y/x]\}\rangle$ (substitution, II)
provided that $x \equiv y$ or $x \notin D$, $y \notin q$

$\langle D \mid \{p\}x:=s\{q\} \Rightarrow \{p[e/y]\}x:=s[e/y]\{q[e/y]\}\rangle$ (substitution, III)
provided $y \not\equiv x$, $y \notin D$, $x \notin e$.

Remark. Note that, in IIa.1, $\langle D \mid \{p[s/x]\}x:=s\{p\}\rangle$ would not work since, in general, $p[s/x]$ is not a well-formed assertion.

THEOREM 5.1. The proof system is sound.

Proof. For the rules of group I this is obvious. For IIa, the assignment axiom follows from $T(p[e/x])(\sigma) = T(p)(\sigma\{E(e)(\sigma)/x\})$, together with $V(e)(\varepsilon)(\sigma) = E(e)(\sigma)$, for ε normal. The recursion rule is a form of Scott's induction (see, e.g., [4]). The composition and conditionals rules are clear. As to group IIb, if $S \rightsquigarrow S'$ is in RS_1, then, by the definition of O_D, we have that $O_D(S) = O_D(S')$, but for the values of $\vec{y}$; hence, for all normal ε, $N_D(S)(\varepsilon) = N_D(S')(\varepsilon)$ (but for ...) and, since the $\vec{y}$ do not occur in p, q, the desired result follows. The rules of group III are partly easy, partly require somewhat tedious manipulations with the substitution lemmas of Section 4 (for related - though not identical - techniques we refer again to De Bakker [4]).

6. COMPLETENESS FOR CALL-BY-VALUE

We consider the language as introduced in Section 2, but restricted by omitting the clause s::=...|<<u>name</u> v:s>|..., i.e., we now only allow call-by-value abstraction, and accordingly restrict the type of all expressions s^τ to $\tau = \omega^n \rightarrow \omega$, $n \geq 0$. The proof system of the previous section - with a few small modifications - can then be shown to be *complete* without too much effort. We first remark that when call-by-value is the only abstraction mechanisms all functions $V(s)(\varepsilon)(\sigma)$ are strict (i.e., $V(s^{\tau})(\varepsilon)(\sigma)(\vec{\alpha}) = \bot_{\tau}$ as soon as any of the α_i equals $\bot_{\omega}$), provided $\varepsilon(g)(\sigma)$ is strict for all free g. We omit the easy proof of this. We now consider the proof system of Section 5, modified as follows:

1. We first replace reduction system RS_1 by RS_{cbv}:
   (i) remove the rule dealing with call-by-name
   (ii) add a rule
   $x:=f\vec{s} \rightsquigarrow (y:=s)^{\rightarrow}; x:=f\vec{y}$, not all $\vec{s}$ simple, where the $\vec{y}$ satisfy the restrictions (i), (ii) and (iii) mentioned before.

   After a corresponding adaptation of the notion of simple statement:

   $T ::= x:=e \mid x:=f\vec{e} \mid T_1;T_2 \mid$ <u>if</u> $e_1=e_2$ <u>then</u> T_1 <u>else</u> T_2 <u>fi</u>

   it can be shown that, with the use of RS_{cbv}, each statement (with only call-by-value) can be reduced to an equivalent (but for the values of the $\vec{y}$) simple statement (see the Appendix).
2. Rule IIa.2 (recursion) is simplified to

   $$\frac{\langle D \mid \{p\}x:=g\vec{e}\{q\} \Rightarrow \{p\}x:=t[g/f]\vec{e}\{q\}\rangle}{\langle D \mid \{p\}x:=f\vec{e}\{q\}\rangle}, \quad g \notin D$$

   Also, rules IIb now refer to the reduction system RS_{cbv}.

Before we can state the completeness theorem, we have to introduce the usual expressibility notion: The interpretation J is *expressive* with respect to the languages *Prog* and *Assn*, provided that for each closed program $P \equiv \langle D \mid S\rangle$, each p, and each normal ε the following holds:

1. There exists an assertion q (the *weakest precondition* wp(P,p)) such that, for all σ,
   $T(q)(\sigma) \iff \forall\sigma'[\sigma'=N(P)(\varepsilon)(\sigma) \wedge \sigma' \neq \bot_{\Sigma} \Rightarrow T(p)(\sigma')]$
2. There exists an assertion r (the *strongest postcondition* sp(p,P)) such that, for all σ,
   $T(r)(\sigma) \iff \exists\sigma'[T(p)(\sigma') \wedge \sigma=N(P)(\varepsilon)(\sigma')]$

(By a remark of Olderog, it is actually sufficient to postulate either 1 or 2.)

The following lemma on sp will be needed below:

LEMMA 6.1. For all σ

a. $T(sp(p,\langle D \mid x:=s\rangle)[y/x])(\sigma) = T(sp(p[y/x],\langle D \mid y:=s[y/x]\rangle)(\sigma)$,
   provided that $x \equiv y$ or $x \notin D$, $y \notin p,D,s$.
b. $T(sp(p,\langle D \mid x:=s\rangle)[e/y])(\sigma) = T(sp(p[e/y],\langle D \mid x:=s[e/y]\rangle)(\sigma)$,
   provided that $y \not\equiv x$, $y \notin D$, $x \notin e$.

Now let us extend the proof system described above with all formulae $\langle D \mid p\rangle$ such that $\models_J \langle D \mid p\rangle$ (i.e., all J-valid assertions are taken as axioms), and let $\vdash_J$ denote provability in this extended system. We then have

THEOREM 6.2. (completeness theorem for call-by-value) For each closed $\langle D|S\rangle$, and expressive J, if $\models_J \langle D|\{p\}S\{q\}\rangle$ then $\vdash_J \langle D|\{p\}S\{q\}\rangle$.

The proof uses the notion of most general formula of Gorelick [11]. Let $D \equiv f \Leftarrow t$, let $u, u_0, \vec{u}_{1:n}$ be different ground variables not appearing in D, and let $r(u,u_0,\vec{u}) \overset{df.}{\equiv}$ sp $(u=u_0, \langle D|u:=f\vec{u}\rangle)$. Next, let $F_0 \overset{df.}{\equiv} \{u=u_0\}u:=g\vec{u}\{r(u,u_0,\vec{u})\}$ for some arbitrary g. We first assert the

LEMMA 6.3. If $\models \langle D|\{p\}S\{q\}\rangle$, and g arbitrary, then $\vdash \langle D|F_0 \Rightarrow \{p\}S[g/f]\{q\}\rangle$.

Proof. Assume $\models \langle D|\{p\}S\{q\}\rangle$.

a. By Section 3 and the definition of the proof system there exists some simple T such that $\models \langle D|\{p\}T\{q\}\rangle$ and $\vdash \langle D|\{p\}T[g/f]\{q\} \Rightarrow \{p\}S[g/f]\{q\}\rangle$.

b. We now prove that (*): for all simple T, if $\models \langle D|\{p\}T\{q\}\rangle$ then $\vdash \langle D|F_0 \Rightarrow \{p\}T[g/f]\{q\}\rangle$. Together with part a, this will establish the desired result. The proof of (*) is by induction the complexity of T. If T is not of the form $x:=f\vec{e}$, the result is easy. Otherwise, we follow the argument as described in [1,4]. □

We now finish the proof of the completeness theorem as follows: By the definition of $r(u,u_0,\vec{u})$ we have $\models \langle D|\{u=u_0\}u:=f\vec{u}\{r(u,u_0,\vec{u})\}\rangle$. Hence, by the fixed point property, $\models \langle D|\{u=u_0\}u:=t\vec{u}\{r(u,u_0,\vec{u})\}\rangle$. By the lemma, we obtain that $\vdash \langle D|F_0 \Rightarrow \{u=u_0\}u:=t[g/f]\vec{u}\{r(u,u_0,\vec{u})\}\rangle$, and, taking $g \not\equiv f$, the recursion rule yields that (**): $\vdash \langle D|F_0\rangle$. By the lemma, assuming $\models \langle D|\{p\}S\{q\}\rangle$, and taking $g \equiv f$, we also obtain $\vdash \langle D|F_0 \Rightarrow \{p\}S\{q\}\rangle$. Together with (**) this yields the desired result $\vdash \langle D|\{p\}S\{q\}\rangle$.

7. THE COMPLETENESS PROBLEM FOR GENERAL ABSTRACTION

We have not been able to find a completeness proof for general abstraction (see also the remarks at the end of this section). However, if we allow both forms of abstraction but restrict *all* types to $\omega^n \to \omega$ $(n \geq 0)$, then we do have a complete system, albeit at the cost of an extension of the language *of the proof system* (i.e., not of the original programming language). The extension is twofold:

1. We introduce an auxiliary assignment statement $x \leftarrow s$, which may be viewed as "non-strict assignment". We allow the modification $\sigma\{\alpha/x\}$ for *any* $\alpha \in V_\omega$ (contrary to the previous situation where $\alpha \neq \bot_\omega$ was required), and put $M(x\leftarrow s)(\varepsilon)(\sigma) = \sigma\{V(s)(\varepsilon)(\sigma)/x\}$. Note that even if evaluation of s does not terminate, a "normal" state $(\neq \bot_\omega)$ is delivered (albeit that its value in x equals $\bot_\omega$). Only if x is used subsequently, nontermination of s is observable. Thus, a natural *operational* semantics of this type of assignment seems not feasible. "$\leftarrow$" will be used below to deal with call-by-name (non-strict abstraction), and ":=" to deal with call-by-value (strict abstraction).
2. The class of simple expressions is extended with $e ::= \ldots|\Omega^\omega$; also, for the assertion $p \equiv (e \neq \Omega^\omega)$ we introduce the notation $e\downarrow$. The valuation E is now extended to

E: *Sexp* $\to (\Sigma \to_s V)$ (instead of *Sexp* $\to (\Sigma_0 \to V_0)$, as before). Possible evaluations now are $E(x)(\sigma\{\perp_\omega/x\}) = \perp_\omega$, and $E(y)(\sigma\{\perp_\omega/x\}) = \sigma(y)$ for $x \not\equiv y$.

The proof system is modified in the following way:

1. Statements S are as before (but with all s of type $\omega^n \to \omega$).
2. *Intermediate* statements R are defined by
   $R ::= x{:=}e \mid x{\leftarrow}d \mid R_1;R_2 \mid$ <u>if</u> $e_1{=}e_2$ <u>then</u> R_1 <u>else</u> R_2 <u>fi</u>
   Here d is auxiliary construct defined by
   $d ::= e \mid f\vec{e} \mid (R;d)$
3. *Simple* statements T are now defined by
   $T ::= x{:=}e \mid x{\leftarrow}e \mid x{\leftarrow}f\vec{e} \mid T_1;T_2 \mid$ <u>if</u> $e_1{=}e_2$ <u>then</u> T_1 <u>else</u> T_2 <u>fi</u>

Moreover, we introduce two reduction systems RS_2 and RS_3:

RS_2 has rules to simplify S to R:

$x{:=}s \leadsto y{\leftarrow}s;\ x{:=}y$	s not simple
$x{\leftarrow}c\vec{s} \leadsto x \leftarrow ((y{:=}s)^{\to};c\vec{y})$	not all $\vec{s}$ simple
$x{\leftarrow}f\vec{s} \leadsto x \leftarrow ((y{\leftarrow}s)^{\to};f\vec{y})$	not all $\vec{s}$ simple
$x \leftarrow$ <<u>val</u> $x':s_0>\vec{s} \leadsto x \leftarrow (y{:=}s_1;\ s_0[y/x']\vec{s}_{2:n})$	
$x \leftarrow$ <<u>name</u> $x':s_0>\vec{s} \leadsto x \leftarrow (y{\leftarrow}s_1;\ s_0[y/x']\vec{s}_{2:n})$	
$x \leftarrow$ <u>if</u> b <u>then</u> s' <u>else</u> s" <u>fi</u> $\vec{s} \leadsto$ <u>if</u> b <u>then</u> $x{\leftarrow}s'\vec{s}$ <u>else</u> $x{\leftarrow}s''\vec{s}$ <u>fi</u>	
(rules 5, 6, 7 of RS_1)	
<u>if</u> $s_1{=}s_2$ <u>then</u> S_1 <u>else</u> S_2 <u>fi</u> $\leadsto y_1{\leftarrow}s_1;\ y_2{\leftarrow}s_2;$ <u>if</u> $y_1{=}y_2$ <u>then</u> S_1 <u>else</u> S_2 <u>fi</u>	not all $\vec{s}$ simple

(As before, the $\vec{y}$ are assumed to be fresh and not in D.)

RS_3 has rules to simplify R to T:

$x \leftarrow (y{\leftarrow}d;d') \leadsto y{\leftarrow}d;\ x{\leftarrow}d'$

$x \leftarrow (y{:=}e;d') \leadsto$ <u>if</u> $e{\downarrow}$ <u>then</u> $y{\leftarrow}e;\ x{\leftarrow}d'$ <u>else</u> $x{\leftarrow}\Omega$ <u>fi</u>

$x \leftarrow ((R_1;R_2);d) \leadsto x \leftarrow (R_1;(R_2;d))$

$x \leftarrow$ (<u>if</u> $e_1{=}e_2$ <u>then</u> R_1 <u>else</u> R_2 <u>fi</u>; d) $\leadsto$
    <u>if</u> $e_1{=}e_2$ <u>then</u> $x \leftarrow (R_1;d)$ <u>else</u> $x \leftarrow (R_2;d)$ <u>fi</u>

Remark. The complications in the systems RS_2, RS_3 are caused by the following phenomenon: The simplification rule

$$x \leftarrow \underline{<\text{val}}\ x':s_0>\vec{s} \leadsto y{:=}s_1;\ x \leftarrow s_0[y/x']\vec{s}_{2:n}$$

is not sound, since the right-hand side might (for nonterminating s_1) transform σ to $\perp_\Sigma$ whereas the left-hand side would yield $\sigma\{\perp_\omega/x\}$. This implies that the assignment $y{:=}s_1$ has to be executed only if x is evaluated, and this motivates the introduction of the intermediate d which are first accumulated and then essentially dealt with through the "$e{\downarrow}$" test in rule RS_3, #2.

LEMMA 7.1. Reduction systems RS_2 and RS_3 always terminate, and yield for each S an equivalent (but for the auxiliary variables) intermediate R, and for each R an equivalent T.

The new proof system now has the following rules:

I' As I (in Section 5).

II' a. $\langle D \mid \{p[e/x]\} x \leftarrow e \{p\} \rangle$

$\langle D \mid \{e\downarrow \supset p[e/x]\} x := e \{p\} \rangle$

$$\frac{\langle D \mid \{p\} x \leftarrow g\vec{e} \{q\} \Rightarrow \{p\} x \leftarrow t[g/f]\vec{e} \{q\} \rangle}{\langle D \mid \{p\} x \leftarrow f\vec{e} \{q\} \rangle} \quad , g \notin D$$

Composition and conditionals as before

b. All rules from $\mathcal{RS}_2$ and $\mathcal{RS}_3$ are turned into proof rules (in the same manner as was done for $\mathcal{RS}_1$ in Section 5).

III' Obtained from III by replacing everywhere ":=" by "$\leftarrow$".

An interpretation $\mathcal{J}$ is called expressive with respect to *Prog* and *Assn*' in the usual way, but observe that *Assn*' now contains assertions involving simple expressions including Ω.

THEOREM 7.2. (soundness and completeness). Let $\vdash_{\mathcal{J}}$ be defined as before.

a. For all $\mathcal{J}$, $\vdash_{\mathcal{J}} \langle D \mid \{p\} S \{q\} \rangle \Rightarrow \models_{\mathcal{J}} \langle D \mid \{p\} S \{q\} \rangle$

b. For expressive $\mathcal{J}$, $\models_{\mathcal{J}} \langle D \mid \{p\} S \{q\} \rangle \Rightarrow \vdash_{\mathcal{J}} \langle D \mid \{p\} S \{q\} \rangle$.

Proof. Similar to that of Theorem 6.2, using the first two rules of II'a to deal with the two forms of assignment. □

Remark. We do not know whether a complete proof system exists for the case of arbitrary types. By an argument as used in Clarke [7], if we could prove the undecidability of the halting problem for programs in our language interpreted over some *finite* domain, then we could infer incompleteness. However, no such undecidability result is available at present. (Neither do we know whether our language allows an application of Lipton's theorem [18].) It seems rather likely that, as soon as we would extend the language with function procedures *with* side-effects (essentially by extending the syntax of expressions with the clause s::=...|S;s and extending $\mathcal{RS}_1$ with the rule x:=S;s → S;x:=s) then Clarke's simulation argument (using an idea of Jones and Muchnik [15]) could indeed be used to obtain undecidability, thus yielding incompleteness.

APPENDIX TERMINATION OF THE REDUCTION SYSTEMS $\mathcal{RS}_1$, $\mathcal{RS}_2$, $\mathcal{RS}_3$.

Ad $\mathcal{RS}_1$ (see Section 3).

We will describe a proof that every statement can be simplified, using these rules (which as always may be applied inside a 'context'), to a *simple* statement, defined as in Section 3; in such a statement none of the simplification rules can be applied. It is only shown that '*innermost*' simplification always terminates; but in fact one can show that *all* simplifications must terminate (even in a unique result). The proof that innermost simplifications must terminate, is in two parts.

The first part is as follows: assign to every 'redex' statement R (i.e., a statement as in the LHS of the simplification rules) a norm $\{R\} \in \mathbb{N}$ such that the newly created redex statements R' in the RHS of the rules have a *smaller* norm. (The norm of redex statements occurring in S_1, S_2 (as displayed in the rules) does not change during the simplification step.) Then assign to an arbitrary statement S which one wants to simplify, the norm $\{\{S\}\} = \langle\{R_1\},\dots,\{R_n\}\rangle$, the 'multi-set' of the norms of all the occurrences of redex statements in S. Now it is easy to see that for an *innermost* simplification step $S \rightsquigarrow S'$ we have $\{\{S\}\} \succ \{\{S'\}\}$, where '$\succ$' is the well-ordering of multisets of natural numbers. Hence every sequence of innermost simplification steps terminates.

The second and more problematic part is to define {R}. This is done by defining $\{x{:=}s\} = \|s\|$ and $\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\} = \|b\|$, where $\|\ \|: \mathit{Exp} \cup \mathit{Bexp} \to \mathbb{N}$ is a suitable complexity measure (norm) which is to be defined yet. Obviously, we require e.g.:

(1) $\|\langle\underline{\text{name}}\ v{:}s\rangle\vec{s}_{1:n}\| > \|s[s_1/v]\vec{s}_{2:n}\|$;

(2) $\|\underline{\text{if}}\ b\ \underline{\text{then}}\ s'\ \underline{\text{else}}\ s''\ \underline{\text{fi}}\ \vec{s}\| > \|b\|,\ \|s'\vec{s}\|,\ \|s''\vec{s}\|$;

(3) $\|\neg b\| > \|b\|$; $\|b_1 \supset b_2\| > \|b_1\|,\ \|b_2\|$,

to name some of the more important requirements. We will define $\|s\|$ and $\|b\|$ by means of the auxiliary reduction system having as set of 'terms' *Exp* ∪ *Bexp* and as 'reduction' rules:

(i) (λ-reduction) $\langle\underline{\text{name}}\ v{:}s\rangle(s_1) \rightsquigarrow s[s_1/v]$
$\langle\underline{\text{val}}\ x{:}s\rangle(s_1) \rightsquigarrow s[s_1/x]$

(ii) (parallel reduction) $\underline{\text{if}}\ b\ \underline{\text{then}}\ s_1\ \underline{\text{else}}\ s_2\ \underline{\text{fi}} \rightsquigarrow s_1$
$\underline{\text{if}}\ b\ \underline{\text{then}}\ s_1\ \underline{\text{else}}\ s_2\ \underline{\text{fi}} \rightsquigarrow s_2$

(These rules may be applied inside a 'context'.)

We claim that every reduction in this auxiliary reduction system terminates. Now for $a \in \mathit{Exp} \cup \mathit{Bexp}$, we define: $\|a\| = \Sigma_{a \twoheadrightarrow a'} |a'|$, where $\twoheadrightarrow$ is the transitive reflexive closure of $\rightsquigarrow$, and $|a'|$ is the length of symbols of a' (counting free variables less than other symbols, for a minor technical reason). The effect is that if $a \rightsquigarrow a'$ then $\|a\| > \|a'\|$, hence we obtain (1) and part of (2) above; and (3) and the remaining part of (2) are obtained since if a is a proper subterm of a', then $\|a\| < \|a'\|$, as the definition of $\|a\|$ readily yields.

Of course, $\|a\|$ is only well-defined as a natural number if there are no infinite reductions $a \rightsquigarrow a' \rightsquigarrow a'' \rightsquigarrow \dots$, i.e., if our claim holds. To establish this *strong* termination property (i.e. *every* reduction sequence in the auxiliary reduction system terminates) constitutes the main problem. A proof of this property is given by the elegant and powerful method of *computability*, which is often used in logic (Proof Theory) to obtain termination results. The method was developed by Tait [22], and independently by some other authors; for more references and some applications, see Troelstra [23].

The termination of RS_{cbv} (see Section 6) follows by the same arguments as used for RS_1.

Ad $\mathcal{RS}_2$ (see Section 7).
Call the LHS of a simplification rule of $\mathcal{RS}_2$ an A-*redex* if it is an assignment $x \leftarrow s$ or $x := s$, and a B-*redex* if it is a conditional statement <u>if</u> b <u>then</u> S_1 <u>else</u> S_2 <u>fi</u>. Note that an A-redex may 'create' a B-redex, and vice versa.

We will measure the complexity of an A-redex by that of s, and of a B-redex by that of the boolean b. So $\{x \leftarrow s\} = \{x := s\} = |s|$ and $\{$<u>if</u> b <u>then</u> S_1 <u>else</u> S_2 <u>fi</u>$\} = |b|$ where $|\ |$ denotes the length in symbols.

Now if S is a statement to be simplified by $\mathcal{RS}_2$, define $\{\{S\}\} = \langle\{R_i\} \mid$ all occurrences of redexes R_i in S$\rangle$. (Here $\langle\ \rangle$ denotes a multiset.) Then it is easy to see that *innermost* simplifications let $\{\{S\}\}$ decrease; hence they must terminate.

One can also show that *all* simplifications in $\mathcal{RS}_2$ terminate, by recognizing $\mathcal{RS}_2$ as a 'regular non-erasing' reduction system in the sense of Klop [16], for which 'weak' and 'strong' termination are equivalent. An alternative, more direct method would be the construction of a more elaborate counting argument.

Ad $\mathcal{RS}_3$ (see Section 7).
Define $|d|_\ell$ as the 'length' of a construct d such that association to the left (w.r.t.;) counts heavier, and assign to $x \leftarrow d$ the norm $\{x \leftarrow d\} = |d|_\ell$. Termination of $\mathcal{RS}_3$ is now easy to prove.

REFERENCES

1. APT, K.R., *Ten years of Hoare's logic, a survey*, in Proc. 5th Scandinavian Logic Symposium (F.V. Jensen, B.H. Mayoh, K.K. Møller, eds.), pp 1-44, Aalborg University Press, 1979 (revised version to appear in ACM TOPLAS).

2. ASHCROFT, E.A., M. CLINT & C.A.R. HOARE, *Remarks on program proving: jumps and functions*, Acta Informatica, <u>6</u>, p. 317, 1976.

3. DE BAKKER, J.W., *Least fixed points revisited*, Theoretical Computer Science, <u>2</u>, pp. 155-181, 1976.

4. DE BAKKER, J.W., *Mathematical Theory of Program Correctness*, Prentice-Hall International, 1980.

5. BARENDREGT, H.P., *The Lambda Calculus, its Syntax and Semantics*, North-Holland, 1981.

6. DE BRUIN, A., *On the existence of Cook semantics*, Report IW 163/81, Mathematisch Centrum, 1981.

7. CLARKE, E.M., *Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems*, J. ACM, <u>26</u>, pp. 129-147, 1979.

8. CLINT, M. & C.A.R. HOARE, *Program proving: jumps and functions*, Acta Informatica, <u>1</u>, pp. 214-224, 1972.

9. COOK, S.A., *Soundness and completeness of an axiom system for program verification*, SIAM J. on Comp., 7, pp. 70-90, 1978.

10. GORDON, M., R. MILNER & C. WADSWORTH, *Edinburgh LCF*, Lecture Notes in Computer Science 78, Springer, 1979.

11. GORELICK, G.A., *A complete axiomatic system for proving assertions about recursive and non-recursive programs*, Technical Report 75, Dept. of Comp. Science, University of Toronto, 1975.

12. HENNESSY, M.C.B., *The semantics of call-by-value and call-by-name in a nondeterministic environment*, SIAM J. on Comp., 9, pp. 67-84, 1980.

13. HENNESSY, M.C.B. & E.A. ASHCROFT, *A mathematical semantics for a nondeterministic typed lambda calculus*, Theoretical Comp. Science, 11, pp. 227-246, 1980.

14. HOARE, C.A.R., *An axiomatic basis for computer programming*, CACM, 12, pp. 576-580, 1969.

15. JONES, N.D. & S.S. MUCHNIK, *Even simple programs are hard to analyze*, JACM, 24, pp. 338-350, 1977.

16. KLOP, J.W., *Combinatory Reduction Systems*, Mathematical Centre Tracts 127, Mathematisch Centrum, 1980.

17. LANGMAACK, H. & E.R. OLDEROG. *Present-day Hoare-like systems for programming languages with procedures: power, limits, and most likely extensions*, in Proc. 7th Coll. Automata, Languages and Programming (J.W. de Bakker & J. van Leeuwen, eds), Lecture Notes in Computer Scence 85, Springer, 1980.

18. LIPTON, R.J., *A necessary and sufficient condition for the existence of Hoare logics*, in Proc. IEEE Symposium Foundations of Computer Science, pp. 1-6, 1977.

19. O'DONNELL, M., *A critique on the foundations of Hoare-style programming logics*, Technical Report, Purdue University, 1980.

20. PLOTKIN, G.D., LCF *considered as a programming language*, Theoretical Comp. Science, 5, pp. 223-256, 1977.

21. REYNOLDS, J.C., *On the relation between direct and continuation semantics*, in Proc. 2nd Coll. Automata, Languages and Programming (J. Loeckx, ed.), pp. 141-156, Lecture Notes in Computer Science 14, Springer, 1974.

22. TAIT, W.W., *Intentional interpretation of functionals of finite type I*, J. Symbolic Logic, 32, pp. 198-212, 1967.

23. TROELSTRA, A.S. et al., *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Lect. Notes in Mathematics 344, Springer, 1973.