PROGRAMMING LANGUAGE SEMANTICS USING EXTENSIONAL λ-CALCULUS MODELS

Herbert Egli

TR 74-206

April 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

PROGRAMMING LANGUAGE SEMANTICS USING
EXTENSIONAL λ-CALCULUS MODELS

Herbert Egli

Abstract:

We prove a theorem which provides an intuitive understanding of the meaning of λ-terms in Scott's extensional λ-calculus models. This allows us to use those models for the definition of high-level programming languages. In order to illustrate this we define a programming language which includes blocks and (arbitrary recursive) procedures. Two aspects justify this approach. First of all, the logical properties of those (typeless) λ-calculus models are appealing for a formalization similar to LCF [6] which formalizes typed λ-calculus models. Secondly, not having the type restrictions that LCF imposes allows us to define the semantics of high-level programming languages in the spirit of "mathematical semantics" [12] (which is usually based on recursively defined domains), thus the semantic nature of syntactic constructs can be exhibited clearly.

Keywords:

Mathematical semantics, Programming Languages, λ-calculus Models, typed, typeless, extensional, Logic, LCF.

Table of Contents

0. Introduction

The various approaches to define the semantics of programming languages are sometimes classified into axiomatic methods [2], operational methods [3],[17] and mathematical methods [10],[12]. Approaches based on ideas of D. Scott fall into the latter category and lie somewhere in between Hoare's axiomatic semantic definitions and definitions using an interpreter. Scott's idea was to define the semantics abstractly, so that it does not depend on any particular machine. The advantage of this is that the semantic nature of a language is exhibited clearly and proofs about semantic aspects can be carried out in a mathematical environment, thus avoiding usually cumbersome comparisions between the effects of running two programs which is necessary in an operational semantics. Furthermore, the semantic functions reflect the underlying idea of how program evaluation is conceived. These properties make Scott's approach well suited for programming language design as illustrated in [14], [15].

The applications just mentioned make use of recursively defined domains whose existence is guaranteed in Scott's theory,[8]. Using those (typeless) domains for semantic descriptions allows the language designer to define the meaning of syntactic constructs to be what they are mathematically. For instance the meaning of a procedure can be defined to be the function (between the appropriate domains) that the procedure actually denotes. This illustrates what we meant above when we said that the semantic nature can be exhibited clearly in Scott's approach. The other advantage that we pointed out — the possibility of formalizing proofs about semantic definitions — has been demonstrated by Milner [7]. He developed LCF, a logic for computable functions [6], in which the semantics of programming languages can be defined and proofs can be carried out using the relatively few rules of the logic. Unfortunately, the restriction to typed functions in LCF results in the loss of the nice properties that we just mentioned for semantic definitions which use recursively defined domains, since those properties can be obtained only if we work over typeless semantic domains. But there is an obvious way to delete the type restrictions in LCF. Using as models the extensional λ-calculus models as discussed in [1], the rules of LCF need only a few modifications in order to formalize properties about those typeless models. The only question then is whether we can use the semantics of extensional models to express what we want. This question arises because it is not obvious that the typeless semantics gives us "what we would expect".

It is clear that we cannot formally prove a theorem of the form "the semantics in the typeless model is what we expect", since we have no precise formalization of the phrase "what we expect". However, if we propound a thesis which offers a precise interpretation to this phrase, then we can examine the question mathematically. What we can assert precisely then is simply stated the following: "Whenever we forget about the types in a typed term and interpret it in the typeless model, we can retrieve the typed meaning from its typeless interpretation." This is the main theorem proved in section 2. We then apply this theorem to a quite specific example in section 3. In order to show that the typeless semantic definition of a language which allows blocks and procedures is "what we expect", we have to analyze the way in which we think about the definition of such a language. We will exhibit clearly the typed meaning behind it which we identify with the intended meaning. Thus the main theorem can be applied to show that the typeless semantics gives us really what we want.

In section 1 we summarize briefly the definitions and results of [1] as far as they are needed in this paper. The theorem which relates the meaning of typed and typeless terms is proved in section 2. Section 3 then illustrates that semantic definitions using extensional λ-calculus models can be written in a natural way. The main theorem is used to show that the typeless definition is correct.

1. Review of typed and typeless languages

We summarize briefly the definitions and properties of typed and typeless languages and their semantics. For more details we refer to [1].

1.1 Complete partially ordered sets

Our semantic domains are complete partially ordered sets (cpo's), i.e. partially ordered sets with the property that

(i) each ascending chain has a least upper bound

(ii) there is a least element, denoted by $\perp$ (bottom).

— The appropriate functions between cpo's D and D' are the functions that respect least upper bounds of chains. They are called continuous functions. The set of all continuous functions from D to D' is itself a cpo under the pointwise induced partial ordering and is denoted by [D,D'].

— The product of any number of cpo's is obtained by taking the cartesian product of the underlying sets with the componentwise induced partial ordering.

— Continuous functions with several arguments can be viewed as functions on iterated function spaces since there is a continuous natural isomorphism $[D \times D', D''] \cong [D,[D',D'']]$ for any cpo's D, D' and D".

— Each continuous function $\Phi \in [D,D]$ has a least fixed point $f \in D$, given by $f = \bigsqcup \Phi^{(k)}(\perp)$, where $\Phi^{(k)}$ means k-times iterated application of Φ.

— The function that assigns to each $\Phi \in [D,D]$ its least fixedpoint in D is continuous and is called the least fixed point operator (for D).

1.2 Typed languages

Types are defined inductively by

— 0 is a type

— if τ,σ are types then $(\tau\to\sigma)$ is a type

These are all the types.

Notation: (i) $\tau\to\sigma\to\rho$ means $(\tau\to(\sigma\to\rho))$

(ii) A special selection of types are the integer types, defined by $n+1 = (n\to n)$.

By a typed language we mean a typed λ-language with some (non-standard) constants (denoted by $a^\tau, b^\sigma, \ldots$), infinitely many variables for each type (denoted by $x^\tau, y^\sigma, \ldots$) and for each type τ the standard constants L_τ, C_τ and U_τ (of type $(\tau\to\tau)\to\tau$, $0\to\tau\to\tau\to\tau$ and τ respectively). Terms are built up from these atoms by means of typed application and λ-abstraction, i.e.

— a^τ, x^τ are terms of type τ;

— $t^{(\tau\to\sigma)}(s^\tau)$ is a term of type σ;

— $[\lambda x^\tau.t^\sigma]$ is a term of type $(\tau\to\sigma)$.

These are all the terms.

A (standard) model (structure) is determined by

(a) a cpo D_0

(b) two disjoint sets TRUE, FALSE $\subseteq (D_0 - \{\perp\})$ with the property that each contains with an element all greater elements as well.

(c) the meaning of the (non-standard) constants (i.e. $\underline{a}^\tau \in D_\tau$, where $D_{\rho\to\sigma} = [D_\rho, D_\sigma]$).

Notation: We use $\alpha^\tau, \beta^\sigma, \ldots$ as variables ranging over $D_\tau, D_\sigma \ldots$

The meaning of the standard constants is given by:

— $\underline{U}_\tau = \perp_\tau \in D_\tau$

— $\underline{L}_\tau$: $[D_\tau, D_\tau] \to D_\tau$ is the least fixed point operator

— $\underline{C}_\tau$: $D_0 \times D_\tau \times D_\tau \to D_\tau$ is the conditional, i.e.

$$\underline{C}_\tau(\alpha^0)(\beta^\tau)(\gamma^\tau) = \begin{cases} \beta^\tau \text{ if } \alpha^0 \in \text{TRUE} \\ \gamma^\tau \text{ if } \alpha^0 \in \text{FALSE} \\ \perp_\tau \text{ ow.} \end{cases}$$

Each interpretation of the variables (i.e. $\underline{x}^\tau \in D_\tau$) extends in the obvious way to an interpretation of all terms ($\underline{t}^\tau \in D_\tau$).

1.3 Extensional λ-calculus models

The definition and the properties of the extensional λ-calculus model D over a cpo D_0 have been discussed in detail in [1]. The construction is the same as given in [11], except that we use cpo's rather than continuous lattices. All we need to know about these models in this report is summarized below. The name extensional λ-calculus model will be justified in the next subsection.

<u>Proposition</u>:

The extensional λ-calculus model D over a cpo D_0 has the following properties:

(1) D is a cpo. For each type τ we have a pair of continuous functions

$$\begin{tikzcd} D_\tau \arrow[r, bend right, "\iota_\tau"'] & D \arrow[l, bend right, "\pi_\tau"'] \end{tikzcd}$$

such that $\pi_\tau \cdot \iota_\tau = Id_{D_\tau}$ and $\iota_\tau \cdot \pi_\tau \sqsubseteq Id_D$.

(2) D is isomorphic to its own function space ($D \cong [D,D]$), i.e. each element $\alpha \in D$ can be used as a continuous function from D to D and vica-versa.

(3) (i) $\iota_{\tau\to\sigma}(\alpha^{\tau\to\sigma}) = \iota_\sigma \cdot \alpha^{\tau\to\sigma} \cdot \pi_\tau$

(ii) $\pi_{\tau\to\sigma}(\alpha) = \pi_\sigma \cdot \alpha \cdot \iota_\tau$

These equations will be used over and over again.

<u>1.4 Typeless languages</u>

By a <u>typeless language</u> we mean a λ-language with constants. Terms are built up by means of application and λ-abstraction from non-standard constants (denoted by a,b,...),(infinitely many) variables (denoted by x,y,...) and the standard constants L,C,U.

A <u>standard model</u> (structure) is determined by

(a) a cpo D_0

(b) TRUE, FALSE as for typed models

(c) the meaning of non-standard constants

(i.e. $\underline{a} \in D$, where D is the λ-calculus model over D_0).

<u>Notation</u>: We use $\alpha, \beta, \ldots$ as variables ranging over D.

The meaning of the standard constants is given by

$\underline{U} = \perp \in D$

$\underline{L} \in D \cong [[D,D],D]$ is the least fixed point operator

$\underline{C} \in D \cong [D \times D \times D, D]$ is the conditional, i.e.

$$\underline{C}\,(\alpha)(\beta)(\gamma) = \begin{cases} \beta & \text{if } \pi_0\alpha \in \text{TRUE} \\ \gamma & \text{if } \pi_0\alpha \in \text{FALSE} \\ \perp & \text{ow.} \end{cases}$$

As in the typed case, each interpretation of the variables (i.e. $\underline{x} \in D$) extends to an interpretation of all terms ($\underline{t} \in D \cong [D,D] \cong \ldots$). Before saying more about that we introduce some notation.

— $t|^x_s$, where t and s are terms and x is a variable, means substitution, i.e. "replace each free occurrence of x in t by s after suitable changes of bound variables in t such that no free variable in s becomes bound after substitution". Consequently, we identify a priori terms that differ only in the naming of their bound variables (α-converted terms).

— $\underline{t|^x_\alpha}$, where t is a term, x a variable and α an element of D, is used as an abbreviation for $\underline{\underline{t}}$, where $\underline{\underline{\ }}$ is the same interpretation as $\underline{\ \ }$ except that $\underline{\underline{x}} = \alpha$.

Proposition:

Each interpretation of the variables in D (i.e. $\underline{x} \in D$) extends to an interpretation of all terms such that

(i) $\underline{[\lambda x.t(x)]} = \underline{t}$ (provided that x is not free in t)

(ii) $\underline{[\lambda x.t](s)} = \underline{t|^x_s}$.

Proof: Given the meaning of the constants and an interpretation of the variables, the interpretation of the other terms is defined inductively by

— $\underline{t(s)} = \underline{t}(\underline{s})$

— $\underline{[\lambda x.t]}$ = element in D corresponding to the function

$$\alpha \mapsto \underline{t|^x_\alpha}.$$

(i) and (ii) are obviously satisfied.

Remark: This proposition states that the interpretation (which is invariant under α-conversion by definition) is also invariant under η-conversion (i) and β-conversion (ii) of λ-terms, thus D is a model for the λ-calculus. It is extensional since unrestricted η-conversion can be applied, as opposed to other models as given for instance in [16].

2. Relation between typed and typeless terms

We now want to "make typed terms typeless" (we call it transformation) and compare the typed meaning of a term with the meaning of the transformed term in a "corresponding" typeless model.

2.1 Derived languages

We start out with a typed language and derive from it a typeless language as follows.

The derived language is a typeless language s.t. the non-standard constants are in 1-1 correspondence with the nonstandard constants of the typed language. We will sometimes write $\{a^\tau\}$ to denote the typeless constant that corresponds to the typed constant denoted by a^τ.

The semantic domain for the derived language is clearly given by the D_0 and the subsets TRUE,FALSE which determine the typed structure. The meaning of the non-standard constants is given by $\{\underline{a}^\tau\} = \iota_\tau \underline{a}^\tau$.

Even if we are interested only in closed terms, free variables are needed over and over again in inductive proofs. If we transform a typed term with free variables then the transformed term will contain free variables, so its meaning depends on an interpretation of the typeless variables. What we clearly want to do is to interpret a free typeless variable as the image of the interpretation of the typed variable from which it was obtained. To keep track in our meta language of the origin of free variables in transformed terms we extend the notion of typeless terms by adding atoms

In general we cannot expect that $\underline{Tr\ t^{\sigma}} = \iota_{\sigma}\underline{t}^{\sigma}$. A simple example is $\underline{Tr[\lambda x^0.x^0]} = \underline{[\lambda x.x]} \neq \iota_1\underline{[\lambda x^0.x^0]}$. Another example is the factorial on the positive integers which can be defined by either

$$t_1 = L_1[\lambda y^1.[\lambda x^0.C_0(x^0=1)(1)(x^0 * y^1(x^0-1))]]$$

or $$t_2 = L_1[\lambda y^1.[\lambda x^0.C_0(x^0=1)(x^0)(x^0 * y^1(x^0-1))]]$$

It is easy to see that

$$\underline{Tr\ t_1} = \iota_1\underline{t_1} = \iota_1\underline{t_2} \neq \underline{Tr\ t_2}\text{ , where for instance}$$

$$Tr\ t_2 = L[\lambda y.[\lambda x.\ C(x = 1)\ (x)\ (x * y(x - 1))]].$$

We used ◯ here to indicate that the respective names now are names for typeless objects. Once we have decided on the typeless semantics we will clearly prefer a simpler notation, for instance

$$L\ \lambda fx.(x = 1 \to x,\ x * f(x-1))$$

or even $f(x) \Leftarrow (x = 1 \to x,\ x * f(x-1))$.

But first we have to show that we do not lose the intended typed meaning when we use the typeless semantics.

2.3 Main theorem

THEOREM: (I) $\underline{t}^{\tau} = \pi_{\tau}(\underline{Tr\ t^{\tau}})$

(II) $\iota_0\underline{t}^0 = \underline{Tr\ t^0}$

of the form $\{x^\tau\}$, where x^τ is a typed variable.

We want to interpret an $\{x^\tau\}$ with respect to a typed interpretation ($\underline{\{x^\tau\}} = \iota_\tau \underline{x}^\tau$), so the meaning of a typeless term now may depend on a typeless and a typed interpretation.

A term is <u>semi-closed</u> if it contains no free (typeless) variables.

<u>Examples</u>:

$\{x^\tau\}$, $[\lambda x.\ a(x(\{x^\tau\}))]$, ... are semi-closed.

x , $[\lambda y.\ a(x(\{x^\tau\}))]$, ... are not semi-closed.

The meaning of a semi-closed term is clearly independent of any interpretation of the typeless variables. In other words, each typed interpretation extends to an interpretation of semi-closed terms.

2.2 Transformation

We now can make precise what we mean by transforming a typed term into a typeless term.

Tr: {typed terms} $\to$ {semi-closed terms}

is defined by

— $\mathrm{Tr}\ L_\tau = L$; $\mathrm{Tr}\ C_\tau = C$; $\mathrm{Tr}\ U_\tau = U$

— $\mathrm{Tr}\ a^\tau = \{a^\tau\}$ for all non-standard constants a^τ

— $\mathrm{Tr}\ x^\tau = \{x^\tau\}$

— $\mathrm{Tr}(t^{\tau\to\sigma}(s^\tau)) = (\mathrm{Tr}\ t^{\tau\to\sigma})(\mathrm{Tr}\ s^\tau)$

— $\mathrm{Tr}[\lambda x^\tau.t^\sigma] = [\lambda x.(\mathrm{Tr}\ t^\sigma)|^{\{x^\tau\}}_{x}]$ for a new variable x.

We prove this theorem in several steps. A straightforward induction proof does not work because

in general $\pi_{\tau+\sigma}(\alpha)(\pi_\tau\beta) \neq \pi_\sigma(\alpha(\beta))$

and also $\underline{L}_\tau(\pi_{\tau+\tau}\alpha) \neq \pi_\tau(\underline{L}(\alpha))$.

<u>Lemma 1:</u> Each typed term has a (β-η-) normal form.

Actually, each typed term is strongly normalizable. A proof of this fact can be found in [13].

<u>Remark:</u> A normal form (NF) has one of the following shapes:

$$\left.\begin{array}{l} \text{— } a^\tau(NF_1)\ldots(NF_k) \\ \text{— } x^\tau(NF_1)\ldots(NF_k) \end{array}\right\} k \geq 0$$

— $[\lambda x^\tau.NF_1]$

<u>Lemma 2:</u> The theorem is true for L_τ-free terms with no C_τ's for $\tau \neq 0$.

<u>Lemma 2.1:</u> Transformation commutes with conversion.

(Proof: immediate)

<u>Proof of lemma 2:</u> Due to lemma 1 and 2.1 it is sufficient to show lemma 2 for terms in normal form. We do this by induction on the length of normal forms, i.e. we show that the theorem is true for

(1) $t = \xi^\tau(NF_1)\ldots(NF_k)$, provided it is true for $NF_1,\ldots,NF_k$ and ξ^τ is either a variable, a non-standard constant or C_0.

(2) $t = [\lambda x^\tau.NF_1]$, provided it is true for NF_1.

The induction hypothesis is included in (1) for $k = 0$.

Case (1a): $\xi^\tau = C_0$.

- Part (I) of the theorem follows from the equations

$$
\begin{aligned}
& (\pi_{0\to0\to0\to0}\underline{C})\ (\alpha^0)\ (\beta^0)\ (\gamma^0) \\
= \ & (\pi_{0\to0\to0}\ \{\underline{C}(\iota_0\alpha^0)\})\ (\beta^0)\ (\gamma^0) \\
= \ & (\pi_{0\to0}\ \{\underline{C}(\iota_0\alpha^0)(\iota_0\beta^0)\})\ (\gamma^0) \\
= \ & \pi_0\ \{\underline{C}(\iota_0\alpha^0)\ (\iota_0\beta^0)\ (\iota_0\gamma^0)\} \\
= \ & \underline{C}_0\ (\alpha^0)\ (\beta^0)\ (\gamma^0)
\end{aligned}
$$

- Part (II) of the theorem (for k=3) follows from the equation

$$\underline{C}\ (\iota_0\alpha^0)(\iota_0\beta^0)(\iota_0\gamma^0) = \iota_0\{\underline{C}_0(\alpha^0)(\beta^0)(\gamma^0)\}.$$

Case (1b): $\xi^\tau \neq C_0$.

We show by induction on k that

$$\underline{Tr\{\xi^\tau(NF_1)\ldots(NF_k)\}} = \iota_\sigma\{\underline{\xi^\tau(NF_1)\ldots(NF_k)}\}$$

where NF_i has type τ_i and $\tau = \tau_1 \to \tau_2 \to \ldots \to \tau_k \to \sigma$

(i) $k = 0$: immediate

(ii) $\underline{Tr\{\xi^\tau(NF_1)\ldots(NF_{k-1})(NF_k)\}}$

$$
\begin{aligned}
&= \underline{Tr\{\xi^\tau(NF_1)\ldots(NF_{k-1})\}}\ \underline{(Tr\ NF_k)} \\
&= \iota_{\tau_k\to\sigma}\{\underline{\xi^\tau(NF_1)\ldots(NF_{k-1})}\}\ \underline{(Tr\ NF_k)} \\
&= \iota_\sigma\{\underline{\xi^\tau(NF_1)\ldots(NF_{k-1})}(\pi_{\tau_k}\ \underline{Tr\ NF_k})\} \\
&= \iota_\sigma\{\underline{\xi^\tau(NF_1)\ldots(NF_{k-1})(NF_k)}\}
\end{aligned}
$$

Case (2): Since $[\lambda x^{\tau}.NF^{\sigma}]$ can not be of type 0, we have to prove only part (I) of the theorem.

$$\underline{[\lambda x^{\tau}.NF^{\sigma}]}(\alpha^{\tau}) = \underline{NF^{\sigma}|^{x^{\tau}}_{\alpha^{\tau}}} = \pi_{\sigma}\{\underline{(\mathrm{Tr}\ NF^{\sigma})|^{\{x^{\tau}\}}_{\iota_{\tau}\alpha^{\tau}}}\}$$

$$= \pi_{\sigma}\{[\underline{\lambda x.(\mathrm{Tr}\ NF^{\sigma})|^{\{x^{\tau}\}}_{x}}](\iota_{\tau}\alpha^{\tau})\} = \{\pi_{\tau\to\sigma}\ \underline{\mathrm{Tr}[\lambda x^{\tau}.NF^{\sigma}]}\}\ (\alpha^{\tau}).$$

Lemma 3: The theorem is true for L_{τ} - free terms.

Proof: This is an easy generalization from lemma 2. Each C_{τ} can be replaced by a typed λ-term containing only C_0 as a constant, and this replacement is compatible with transformation. These replacements are given inductively by

$$C_{\sigma\to\rho} \sim [\lambda x^{0}.\lambda y^{\sigma\to\rho}.\lambda z^{\sigma\to\rho}.\lambda w^{\sigma}.C_{\rho}(x^{0})(y^{\sigma\to\rho}(w^{\sigma}))(z^{\sigma\to\rho}(w^{\sigma}))]$$

The corresponding transformations are

$$C \text{ and } [\lambda x.\lambda y.\lambda z.\lambda w.\ C(x)(y(w))(z(w))].$$

They have the same meaning in the typeless model.

We now want to include also least fixed point operators. For this we define for each k a type-preserving construction.

$$S_k\colon \{\text{typed terms}\} \to \{L_{\tau} - \text{free typed terms}\}$$

by $S_k L_{\tau} = [\lambda x^{\tau\to\tau}.(x^{\tau\to\tau})^{(k)}(U_{\tau})]$; $[(x^{\tau\to\tau})^{(k)}(U_{\tau})$ means $\underbrace{x^{\tau\to\tau}(\ldots x^{\tau\to\tau}}_{\text{k-times}}(U_{\tau})\ldots)]$

$S_k a^\tau = a^\tau$ for all constants $\neq L_\tau$;

$S_k x^\tau = x^\tau$

$S_k(t^{\tau\to\sigma}(s^\tau)) = (S_k t^{\tau\to\sigma})(S_k s^\tau)$;

$S_k[\lambda x^\tau . t^\sigma] = [\lambda x^\tau . S_k t^\sigma]$.

<u>Lemma 4</u>: (i) $\underline{t}^\tau = \bigsqcup_k \underline{S_k t^\tau}$

(ii) $\underline{\mathrm{Tr}\ t^\tau} = \bigsqcup_k \underline{\mathrm{Tr}\ S_k t^\tau}$

<u>Proof</u>: (i) Straightforward induction.

(ii) To prove this we need a stronger version:

For all typeless interpretations,

$$\underline{\mathrm{Tr}\ t^\tau|^{\{x^\sigma\},\ldots}_{x\ \ ,\ldots}} = \bigsqcup_k \underline{\mathrm{Tr}\ S_k t^\tau|^{\{x^\sigma\},\ldots}_{x\ \ ,\ldots}},$$

where all subterms of the form $\{y^\rho\}$ are replaced by typeless variables.

We prove this by induction on the length of t.

— $t = L_\tau$: $\underline{\mathrm{Tr}\ L_\tau} = \underline{L} = \bigsqcup_k \underline{[\lambda x.x^{(k)}(U)]} = \bigsqcup_k \underline{\mathrm{Tr}\ S_k L_\tau}$.

— $t = a^\tau, x^\tau$: immediate

— $t = s^{\tau\to\sigma}(r^\tau)$:

$$\underline{\mathrm{Tr}\ s^{\tau\to\sigma}(r^\tau)|^{\cdots}_{\cdots}} = (\underline{\mathrm{Tr}\ s^{\tau\to\sigma}|^{\cdots}_{\cdots}})(\underline{\mathrm{Tr}\ r^\tau|^{\cdots}_{\cdots}})$$

$$= (\bigsqcup_k \underline{\mathrm{Tr}\ S_k s^{\tau\to\sigma}|^{\cdots}_{\cdots}})(\bigsqcup_e \underline{\mathrm{Tr}\ S_e r^\tau|^{\cdots}_{\cdots}})$$

$$= \bigsqcup_k \underline{(\mathrm{Tr}\ S_k s^{\tau\to\sigma})(\mathrm{Tr}\ S_k r^\tau)|^{\cdots}_{\cdots}} = \bigsqcup_k \underline{\mathrm{Tr}\ S_k(s^{\tau\to\sigma}(r^\tau))|^{\cdots}_{\cdots}}.$$

— $t = [\lambda x^{\tau}.s^{\sigma}]$:

$$\underline{\mathrm{Tr}[\lambda x^{\tau}.s^{\sigma}]\,|^{\{y^{\rho}\},\dots}_{y\ ,\dots}}\ (\alpha) = \underline{[\lambda x.\ \mathrm{Tr}\ s^{\sigma}\,|^{\{x^{\tau}\},\{y^{\rho}\},\dots}_{x\ \ ,\ y\ \ ,\dots}\ ]}\ (\alpha)$$

$$= \underline{\mathrm{Tr}\ s^{\sigma}\,|^{\{x^{\tau}\},\{y^{\rho}\},\dots}_{x\ \ ,\ y\ \ ,\dots}\,|^{x}_{\alpha}} = \bigsqcup_{k} \underline{(\mathrm{Tr}\ S_k s^{\sigma}\,|^{\{x^{\tau}\},\{y^{\rho}\},\dots}_{x\ \ ,\ y\ \ ,\dots})\,|^{x}_{\alpha}}$$

$$= \bigsqcup_{k} \underline{[\lambda x.\ \mathrm{Tr}\ S_k s^{\sigma}\,|^{\{x^{\tau}\},\{y^{\rho}\},\dots}_{x\ \ ,\ y\ \ ,\dots}]}\ (\alpha) = \bigsqcup_{k} \underline{\mathrm{Tr}[\lambda x^{\tau}.S_k s^{\sigma}\,|^{\{y^{\rho}\},\dots}_{y\ ,\dots}]}\ (\alpha)$$

$$= \bigsqcup_{k} \underline{\mathrm{Tr}\ S_k[\lambda x^{\tau}.s^{\sigma}]\,|^{\{y^{\rho}\},\dots}_{y\ ,\dots}}\ (\alpha)$$

Proof of the theorem:

$$\pi_{\tau}\ \underline{\mathrm{Tr}\ t^{\tau}} = \pi_{\tau} \bigsqcup_{k} \underline{\mathrm{Tr}\ S_k t^{\tau}} = \bigsqcup_{k} \pi_{\tau}\ \underline{\mathrm{Tr}\ S_k t^{\tau}} = \bigsqcup_{k} \underline{S_k t^{\tau}} = \underline{t}^{\tau}$$

Lemma 4 Lemma 3 Lemma 4

$$\underline{\mathrm{Tr}\ t}^{0} = \underline{\mathrm{Tr}\ S_k t^{0}} = \bigsqcup_{k} \iota_0\ \underline{S_k t}^{0} = \iota_0 \bigsqcup_{k} \underline{S_k t}^{0} = \iota_0 \underline{t}^{0}$$

3. Application to programming language semantics

In this section we want to illustrate how the theorem that we just proved allows us to use typeless models for the definition of programming languages. Although our typeless meaning function will not be the transformation of a typed term (otherwise we would not make it typeless), we somehow think in a typed way about it. A closer look at this will show that our main theorem can be applied to make sure that the typeless semantics gives us what we expect.

The language that we are going to define is an Algol-like language with assignment, conditional, while and compound statements, blocks and procedures. As long as we do not include procedures the definitions could be written in LCF. We really make use of the possibility of typeless functions only when we introduce procedures. We will therefore treat that part of the semantic definition in more detail. For the definition of simple expressions, assignment, conditional, while and compound statements we can refer to Milner [5], although our meaning function has to be written using the concepts of "environments" and "stores" rather than just "states" so that we can handle different bindings of the identifiers during the evaluation of a program. How this can be done is outlined in the first subsection.

3.1 Semantic concepts

The model in which we will interpret the semantic definitions is built up over the following basic sets.

SYN: It contains all syntactic entities (related by the abstract syntax). In particular it contains ID (identifiers) and SE (statements and expressions).

\$ID: This is a copy of the identifiers outside SYN. The function $\$:ID \to \$ID$ is used to refer to those copies.

PRIM: It contains primitive values, in particular INT (integers) and BOOL (boolean values).

LOC: This is a set of internal names.

We take D_0 to be the cpo obtained from the disjoint union of the basic sets by adding a bottom. We choose the truth values in BOOL to denote true and false in the model, i.e. the subsets TRUE and FALSE of D_0 are the single-point subsets containing the respective elements of BOOL. In our λ-language we can refer to those truth values by T and F respectively.

<u>Environments</u> and <u>stores</u> are defined as follows.

$ENV = (ID \cup \$ID \to LOC)$:

"An environment is a (partial) function from identifiers (for technical reasons we include also \$ID) to locations." What we actually mean by that is an element in the model which is the least extension of such a partial function.

$STORE = (LOC \to$ "values"$)$:

"A store is a function which assigns to each location its content." Again, what we actually mean is an element of the model.

Remark: As long as we do not have procedures we can think of a store as being a function LOC → PRIM, i.e. a store can be thought of as a function of type 1 for the moment.

The evaluation of an expression or a statement in a program depends on the current environment and the store. The evaluation of an expression returns a primitive value, but it might also change the store. With the convention that statements return $\perp$ as value we can treat statements and expressions (SE) using a single meaning function M which is thought of as a function

$$M: SE \rightarrow ENV \rightarrow STORE \rightarrow (STORE; PRIM).$$

Notation: We use $(x;y)$ as an abbreviation for

$$\lambda k.\ k=1 \rightarrow x, (k=2 \rightarrow y,\ U)$$

Remark: In LCF, x and y in $(x;y)$ would be required to have the same type, so we could not define M as above even if environments and stores were typed. To keep the types straight we could clearly separate M into a component which gives the new store and a component which gives the value of an expression. Our notation is just a matter of convenience and does not affect our claim that all definitions could be given in LCF as long as we do not include procedures. We have to point this out because our justification for the correctness of the typeless definitions will use this fact.

The definitions of M for simple expressions, assignment, conditional, while and compound statements can be given similar to the ones in [5], except that we use the concepts of "environments" and "stores" rather than just "states." Let us give for instance the definition for conditional statements, so that we can illustrate also how we take into account that the evaluation of an expression might change the store. For $\rho \in$ ENV and $\sigma \in$ STORE we define

$$M[\underline{\text{If}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2](\rho)(\sigma) \leftarrow$$

$$M[b](\rho)(\sigma)(2) \rightarrow M[S_1](\rho)(M[b](\rho)(\sigma)(1)),$$
$$M[S_2](\sigma)(M[b](\rho)(\sigma)(1))$$

Let us illustrate also how this would look incorporated into the full definition of M.

$$
\begin{aligned}
M \equiv\ & L\lambda M.\lambda q.\lambda\rho.\lambda\sigma. \\
& \vdots \\
& \text{Iscondst}(q) \rightarrow \\
& \quad \{\lambda m.m(2) \rightarrow M(\text{first-of}(q))(\rho)(m(1)), \\
& \quad\quad\quad\quad\quad\quad M(\text{second-of}(q))(\rho)(m(1))\} \\
& \quad\quad\quad\quad\quad (M(\text{test-of}(q))(\rho)(\sigma)), \\
& \vdots
\end{aligned}
$$

emark: We used λ-abstraction to avoid writing three times (test-of(q))(ρ)(σ). This technique often shortens expressions onsiderably.

3.2 Semantics of blocks and procedures

We first need a mechanism which allows us to find new loca-ions during the evaluation of a program. For this we assume that C is order-isomorphic to the natural numbers with initial location and successor function Next : LOC $\rightarrow$ LOC. In the initial location we always keep the name of the next location not used so far. The whole program will be started in the initial store

$$\sigma_0 = (\lambda n.\ n = n_0 \rightarrow Next(n_0), U).$$

For binding (a list of) identifiers (or their copies respectively) to new locations we define the functions

$$New1,\ New2\colon ID\text{-}LIST \rightarrow ENV \rightarrow STORE \rightarrow (ENV;\ STORE).$$

Using the abbreviation $[\alpha|\beta|\gamma]$ for $(\lambda k.\ k = \beta \rightarrow \gamma,\ \alpha(k))$ these functions are defined by

$$\begin{aligned}
&New1(\ell)(\rho)(\sigma) \Leftarrow\\
&\quad null(\ell) \rightarrow (\rho;\sigma),\\
&\qquad\qquad New1(tl(\ell))([\rho|hd(\ell)|\sigma(n_0)])([\sigma|n_0|Next(\sigma(n_0))])
\end{aligned}$$

$$\begin{aligned}
&New2(\ell)(\rho)(\sigma) \Leftarrow\\
&\quad null(\ell) \rightarrow (\rho;\sigma),\\
&\qquad\qquad New2(tl(\ell))([\rho|\$hd(\ell)|\sigma(n_0)])([\sigma|n_0|Next(\sigma(n_0))])
\end{aligned}$$

A <u>block</u> consists of an identifier list ℓ_1 (for simple variables), an identifier list ℓ_2 (for procedure names) and a statement S. We define its meaning by

$$M[\ell_1, \ell_2, S](\rho)(\sigma) \Leftarrow (M[S]*New2[\ell_2]*New1[\ell_1])(\rho)(\sigma),$$

where $(a*b)(\rho)(\sigma) \Leftarrow a(b(\rho)(\sigma)(1))(b(\rho)(\sigma)(2))$.

This definition is motivated by the following convention for <u>procedures</u>. Any procedure call can be used either as an expression or a statement. Each procedure call returns a primitive value (possibly undefined) in the following way. The procedure name p is viewed as a local variable in the body of the procedure definition, except in the construct "call p(e)" which calls the procedure recursively.

The value of a procedure call is then the value of the local variable p before exiting the procedure body.

For the semantic description we separate the two different "meanings" of the same identifier p by using the copy of p in \$ID each time we refer to the procedure and we use p itself as a simple variable. We require that a location is reserved for \$p in the environment in which the procedure is defined, i.e. p must have been declared to be a procedure name.

The evaluation of a procedure call with actual parameter list $<e_1,\ldots,e_n>$ in an environment ρ_1 and a store σ_1 results in a new store and returns a primitive value (possibly $\perp$). We can express this effect by a function

$$F_q^{(\rho)}: \text{EXPR-LIST} \to \text{ENV} \to \text{STORE} \to (\text{STORE}; \text{PRIM}) ,$$

where q is the procedure definition and ρ is the environment in which q appears. Let us illustrate the definition of $F_q^{(\rho)}$ by restricting ourselves for the moment to procedures with only one parameter which is to be called by value. Let $q = [p,x,S]$ be such a procedure definition. Then $F_q^{(\rho)}$ can be defined by

$$F_q^{(\rho)} \equiv \lambda\varepsilon.\lambda\rho_1.\lambda\sigma_1.(\bar{\sigma};\bar{\sigma}(s(1)[p]))$$

where $\bar{\sigma} \equiv M[S](s(1))([s(2)|s(1)[x]|M(\varepsilon)(\rho_1)(\sigma_1)(2)])(1)$

and $s \equiv \text{Newl}(<x,p>)(\rho)(M(\varepsilon)(\rho_1)(\sigma_1)(1))$

In order to supply $F_q^{(\rho)}$ when it is needed, we can put it in the store at the time the procedure is defined. Thus we define the meaning of a <u>procedure definition</u> q with name p by

$$M[q](\rho)(\sigma) \to ([\sigma|\rho(\$p)|F_q^{(\rho)}]; U).$$

The meaning of a <u>procedure call</u> is then defined by

$$M[\text{call } p\langle e_1,\ldots,e_n\rangle](\rho)(\sigma) \leftarrow \sigma(\rho(\$p))(\langle e_1,\ldots,e_n\rangle)(\rho)(\sigma).$$

We notice that the meaning function M now becomes really typeless since STORE can no longer be interpreted as a typed domain (it contains elements which include functions from STORE to STORE). Nevertheless, in view of the theorem of section 2 it is intuitively quite obvious that the interpretation of M in the typeless model gives us exactly what we expect. We will discuss this in more detail in 3.3, but first we would like to illustrate by another example the definition of $F_q^{(\rho)}$ which we have given above only for a simple kind of procedures. We now want to define $F_q^{(\rho)}$ for procedures with an arbitrary number of arguments which are to be called by reference (other calling mechanisms can be treated accordingly). The $F_q^{(\rho)}$ for such a procedure definition $q = [p, \text{f-list}, S]$ can be defined by

$$F_q^{(\rho)}[\text{a-list}](\rho_1)(\sigma_1) \leftarrow (\bar{\sigma}; \bar{\sigma}(s(1)[p]))$$

where $\bar{\sigma} \equiv (M[S](s(1))(s(2))(1)$ and $s \equiv \Phi[\text{f-list}][\text{a-list}](\rho)(\sigma_1)$

Φ is the function that creates the "state" s (an ENV-STORE-pai: in which S is to be evaluated. It is defined recursively by

$$
\begin{aligned}
&\Phi(f)(a)(\rho')(\sigma') \leftarrow \\
&\quad \text{null}(f) \rightarrow \text{New1}(\langle p\rangle)(\rho')(\sigma'),\\
&\quad \text{null}(a) \rightarrow U,\\
&\quad \text{isid}(\text{hd}(a)) \rightarrow \Phi(\text{tl}(f))(\text{tl}(a))([\rho'|\text{hd}(f)|\rho_1(\text{hd}(a))])(\sigma'),\\
&\quad \text{isexpr}(\text{hd}(a)) \rightarrow\\
&\qquad \Phi(\text{tl}(f))(\text{tl}(a))(\bar{s}(1))([\bar{s}(2)|\bar{s}(1)(\text{hd}(f))|M(\text{hd}(a))(\rho_1)(\sigma')(2)]),\\
&\qquad \text{where } \bar{s} \equiv \text{New 1}(\langle\text{hd}(f)\rangle)(\rho')(M(\text{hd}(a))(\rho_1)(\sigma')(1)).
\end{aligned}
$$

3.3 Correctness of the typeless definition

We have emphasized repeatedly that the semantic definition can be interpreted in a typed model as long as we do not have procedures. We now use this fact to show the correctness of the typeless definition including procedures.

The abstract meaning of a procedure is a function which — depending on the arguments, the environment and the store in which the procedure is called — returns the store that we have after the evaluation of the procedure and the value of this call. If we know this function, then the procedure is completely determined and we do not have to know anything about its syntactic definition (in particular, it does not matter in what language it is defined). It is therefore appropriate to view this function as the value associated with the procedure name. That is precisely what we did when we put this function into the location bound to the corresponding procedure identifier. But as we pointed out, doing this makes the meaning function typeless because STORE becomes a typeless object. However, we could avoid this. If we did not store directly the semantic function of a procedure but an encoding of this function instead (i.e. an element of type 0), then STORE would remain typed. Consequently, the meaning function $\tilde{M}$ that we would obtain could be viewed as the transformation of a typed term. We are now going to use this idea to show the desired connection between the typeless definition M and the intended typed meaning.

A meaning function $\tilde{M}$ can be defined such that it encodes the syntax of a procedure definition and the environment in which this definition is encountered. The semantic function then is built only when the procedure is called. We define the typed interpretation of $\tilde{M}$ to be the correct semantic definition. This has the following consequences:

(1) $\tilde{M}$ describes the correct semantics because it is the transformation of a typed term which expresses what we want (main theorem).

(2) On any program π, M and $\tilde{M}$ have essentially the same effect. The only difference is that different things are stored in locations bound to procedure names. For the other locations it does not matter whether a procedure call retrieves directly the appropriate meaning function from the store (this happens if we use M) or whether it retrieves only an encoding of this function (if we use $\tilde{M}$). This means that the contents of the locations bound to identifiers which are not procedure names change in exactly the same way using either definition. In particular, the output values are therefore the same for both M and $\tilde{M}$. Thus M describes the correct semantics and that is what we wanted to show.

4. Conclusion

We have shown that the typeless semantics given by extensional λ-calculus models preserves the intended typed meaning and that it can therefore be used in a natural way for programming language definitions. We have discussed in some detail the semantic definition of procedures in order to illustrate what we mean by "the intended typed meaning". To give another example let us briefly indicate how we can describe jumps. Adopting the idea of continuations [9],[18] we can store the continuation that has to be applied for a goto-statement in the location of the corresponding label when we enter the innermost block in which the labelled statement appears. The goto-statement then can retrieve this continuation from the store. Here again we make real use of the typeless semantics but we think in a typed way about this definition (we just store the appropriate continuation to supply it automatically when it is needed). Thus the main theorem applies and proves the definition correct.

Using extensional λ-calculus models for the definition of programming languages combines the idea of LCF with the characteristics of definitions using recursively defined domains. We can look at it as a generalization of LCF to a typeless system which allows us to define the semantics of programming languages in the spirit of "mathematical semantics". On the other hand, we can view it as a unification of semantic definitions using recursively defined domains, thus providing a general logical system for such definitions.

With minor modifications we could use the transformed inference rules of LCF for a typeless logic. It is another question however, precisely how the typeless logic should be formulated in order to be appropriate for mechanical proofs. Analyzing first the kind of things that we want to prove about language definitions should help to answer this question.

Acknowledgements:

I am grateful to Robert L. Constable and David Gries for many helpful discussions.

References

[1] Egli, H., "An Analysis of Scott's λ-calculus Models", TR 73-191, Computer Science Department, Cornell University (1973).

[2] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", CACM 12, #10 (Oct. 1969), 576-583.

[3] Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda Notation", CACM 8, #2,3 (Feb., March 1965), 89-101, 158-165.

[4] Milner, R., "Logic for Computable Functions, Description of a Machine Implementation", AIM - 169/CS-288, Computer Science Department, Stanford University (1972).

[5] Milner, R., "Implementation and Applications of Scott's Logic for Computable Functions", Proc. of an ACM Conference on Proving Assertions about Programs, Las Cruces, New Mexico (1972).

[6] Milner, R., "Models of LCF", AIM-186/CS-332, Computer Science Department, Stanford University (1973).

[7] Milner, R. and R. Weyhrauch, "Proving Compiler Correctness in a Mechanized Logic", Machine Intelligence 7, ed. D. Michie, Edinburgh University Press (1972).

[8] Reynolds, J.C., "Notes on a Lattice-Theoretic Approach to the Theory of Computation", Systems and Information Science, Syracuse University (1972).

[9] Reynolds, J.C., "A Justification of Continuations", Syracuse University, preliminary draft (1973).

[10] Scott, D., "Outline of a Mathematical Theory of Computation", Oxford University Computing Laboratory, Technical Monograph PRG-2 (1970).

[11] Scott, D., "Continuous Lattices", Proc. Dalhousie Conference on Toposes, Algebraic Geometry and Logic, Springer Lecture Notes in Mathematics #274, Springer Verlag Berlin (1972).

[12] Scott, D. and C. Strachey, "Toward a Mathematical Semantics for Computer Languages", Proc. of a Symposium on Computer and Automata, New York, Polytechnic Institute of Brooklyn (1971).

[13] Stenlund, S., Combinators, λ-terms and Proof Theory, D. Reidel Publishing Company/Dordrecht-Holland (1972).

[14] Tennent, R.D., "Mathematical Semantics and Design of Programming Languages", Ph.D. Thesis, Department of Computer Science, University of Toronto (Sept. 1973).

[15] Tennent, R.D., "Mathematical Semantics of Snobol 4", Proc. of an ACM Conference on Principles of Programming Languages, Boston (Oct. 1973).

[16] Wadsworth, C.P., "Semantics and Pragmatics of the Lambda-Calculus", Ph.D. Thesis, University of Oxford (1971).

[17] Wegner, P., "The Vienna Definition Language", Computing Surveys 4, #1 (March 1972).

[18] Strachey, C. and C.P. Wadsworth, "Continuations. A Mathematical Semantics for Handling Full Jumps", Oxford University Computing Laboratory, Technical Monograph PRG-11 (1974).