Induction, Domains, Calculi: Strachey's Contributions to Programming-Language Engineering

David A. Schmidt

Computing and Information Sciences Department, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506 USA.[1]

Abstract. In collaboration with his colleagues at Oxford, Christopher Strachey pioneered the analysis of programming languages in terms of semantic features. Three of Strachey's contributions—inductive definition of semantics, semantic-domain definitions, and calculi for semantic description—are presented, and their consequences on languages research are described. Strachey's impact, present and future, on the computer programming "mainstream" is also outlined.

Why aren't new programming languages designed by programming-language experts? Sometimes they are, but the requirements of marketing, user support, and financial backing mean that the visibility and viability of a new programming language depend mostly on non-programming-language issues. Nonetheless, the academic discipline of programming-language design has proven valuable, because many of its insights and techniques move into the mainstream over a period of decades.

In particular, the "first wave" of language-design innovations, that of syntax definition [36], parsing theory [23], and simple data typing [19], developed in the 1960's and 1970's, are firmly implanted within the mainstream today. One no longer needs to justify why a language's syntax is based on an $LALR(k)$-grammar and why its data-type checking relies on a propositional logic of types. Practicing programmers know and readily apply these First Wave concepts in their daily efforts.

Christopher Strachey led a "second wave" of language-design research, where emphasis shifted from syntactic structure to semantic structure[2] [3]. The Second Wave has not fully impacted the mainstream community at the time of this writing, although some preliminary tremors have been felt. (Perhaps several more "software crises" will be required for the full force to hit!)

With his work in denotational semantics, Strachey tried to make languages easier to understand and *to engineer*. Strachey appreciated that no programming language appears in isolation—a language evolves from existing languages, in response to unsatisfied programming problems. Strachey knew that a competent

[1] `schmidt@cis.ksu.edu`. Supported by NASA NAG-2-1209 and NSF CCR-9970679.

[2] By no means, do we wish to underestimate the roles played by Professor Dana Scott and Strachey's Oxford colleagues, who helped Strachey refine his ideas and state them precisely. Scott's contributions, in particular, appear most everywhere in Strachey's final work, and it is difficult and ultimately unfair to attribute any particular contribution to precisely one person.

language designer intuitively employs a "meta-theory" of language design; by making this meta-theory explicit, one inserts solid engineering into the language-design process.

In this paper, we document three Second-Wave contributions in which Strachey was a major participant—inductive definition of semantics, semantic-domain definitions, and calculi for semantic description—and we describe how these advances have generated a "third wave" of research. We conclude by speculating on the impact of these contributions on the programming-language mainstream's future.

1 Three Contributions of Denotational Semantics

A century from now, the impact of denotational semantics upon computing science will be properly appreciated. But even at the time of this writing, it is clear that denotational semantics has affected the computing community—*denotational semantics demonstrated that one can describe concisely and elegantly what a language means and what its programs mean.* Along with axiomatic semantics [10, 18, 20], denotational semantics made clear that a programming language can be studied as an entity in its own right. The "denotational-semantics method" has set standards that present-day language designers must match, regardless of the techniques they use.

Three crucial technical contributions from denotational semantics stand out:

- *inductive definition of semantics*: The meaning of a language construct is defined in terms of the meanings of its subconstructs.
- *semantic-domain definitions*: The values computed by a language's programs are grouped into named sets, in a manner resembling BNF definitions.
- *calculus for semantic description*: A minimal meta-language with a precise semantics encodes (or "programs") the semantics of a language's constructs.

Each of these contributions receives closer scrutiny in the sections that follow.

2 Inductive Definition

Scott and Strachey once described the intent of denotational semantics as "extending BNF to semantics"—a language construct's syntax *and* semantics are expressed in terms of its subconstructs [46]. The classic example of such *inductive definition* is the semantics of arithmetic expressions, displayed in Figure 1. The inductive structure of each BNF rule directs the definition of a *valuation function*, where one valuation equation appears for each syntax construction.[3]

[3]Although inductive definition was already available in the guise of attribute grammars [24], it was Scott and Strachey's adaptation of the form of inductive definition used in symbolic-logic model theory that made the stronger impact.

BNF definition:

$$
\begin{array}{l}
\mathrm{E} \in \text{Expression} \\
\mathrm{N} \in \text{Numeral} \\
\quad \mathrm{E} \ ::= \ \mathrm{E}_1 + \mathrm{E}_2 \ \mid \ \mathrm{N} \\
\quad \mathrm{N} \ ::= \ 0 \ \mid \ \mathit{succ}\,\mathrm{N}
\end{array}
$$

Semantic valuation functions:

$$
\begin{array}{l}
\mathcal{E} : \text{Expression} \rightarrow \mathit{Integer} \\
\quad \mathcal{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = \mathit{plus}(\mathcal{E}[\![\mathrm{E}_1]\!], \mathcal{E}[\![\mathrm{E}_2]\!]) \\
\quad \mathcal{E}[\![\mathrm{N}]\!] = \mathcal{N}[\![\mathrm{N}]\!] \\
\mathcal{N} : \text{Numeral} \rightarrow \mathit{Integer} \\
\quad \mathcal{N}[\![0]\!] = \mathit{0} \\
\quad \mathcal{N}[\![\mathit{succ}\ \mathrm{N}]\!] = \mathit{plus}(\mathit{1}, \mathcal{N}[\![\mathrm{N}]\!])
\end{array}
$$

Figure 1: Syntax and semantics of arithmetic

$$
\begin{array}{l}
\mathcal{E}[\![\mathit{succ}\ 0 + (0 + \mathit{succ}\ 0)]\!] \\
= \mathit{plus}(\mathcal{E}[\![\mathit{succ}\ 0]\!], \mathcal{E}[\![0 + \mathit{succ}\ 0]\!]) \\
= \mathit{plus}(\mathcal{N}[\![\mathit{succ}\ 0]\!], \mathit{plus}(\mathcal{E}[\![0]\!], \mathcal{E}[\![\mathit{succ}\ 0]\!])) \\
= \mathit{plus}(\mathit{plus}(\mathit{1}, \mathcal{N}[\![0]\!]), \mathit{plus}(\mathcal{N}[\![0]\!], \mathcal{N}[\![\mathit{succ}\ 0]\!])) \\
= \mathit{plus}(\mathit{plus}(\mathit{1}, \mathit{0}), \mathit{plus}(\mathit{0}, \mathit{plus}(\mathit{1}, \mathcal{N}[\![0]\!]))) \\
= \mathit{plus}(\mathit{1}, \mathit{plus}(\mathit{0}, \mathit{plus}(\mathit{1}, \mathit{0}))) \\
= \mathit{2}
\end{array}
$$

Figure 2: Calculating the semantics of an arithmetic expression

An immediate benefit of inductive definition is its elegant application by means of "semantics through substitution"; see Figure 2. Such substitutions train a programmer to understand a language in terms of the source language constructs themselves rather than the underlying sequences of machine operations to which the constructs might translate.

Another crucial advantage of inductive definition is that proofs of language properties can be conducted by induction on the structure of the language's BNF definition. For example, one can readily prove that all Expression syntax phrases in Figure 1 have meanings that are nonnegative integers.

Finally, the inductive-definition pattern gives a language designer a template for language engineering: Each syntactical construct *must* receive a semantic equation, where the semantics of the subconstructs are specified "automatically"—inductively.

It is tempting to violate the inductive engineering approach at the crucial moment of defining the semantics of an iterative construct, e.g., `while` E `do` C `od`, but by staying true to form, one is forced to understand iteration in terms of

Fortran-like:	Lisp-like:
Elementary domains:	*Elementary domains:*
Integer (integers)	*Atom* (atomic values)
Location (Storage locations)	*Nil* (nil value)
Derived domains:	*Derived domains:*
$Environment = \text{Identifier} \rightarrow Denotable$	$Environment = \text{Identifier} \rightarrow Denotable$
$Store = Location \rightarrow Storable$	$Function = Environment \rightarrow Expressible \rightarrow Expressible$
$Procedure = Store \rightarrow Store$	$List = Nil + (Atom \times List)$
Characteristic domains:	*Characteristic domains:*
$Denotable = Location + Procedure$	$Expressible = Atom + List + Function$
$Expressible = Integer$	$Denotable = Expressible$
$Storable = Integer$	

Figure 3: Semantic domains for two languages

arguably more fundamental notions:

$$\mathcal{C}[\![\texttt{while}\ \mathrm{E}\ \texttt{do}\ \mathrm{C}\ \texttt{od}]\!]\rho\sigma = f\sigma$$
$$\text{whererec } f\sigma' = \mathit{if}\ \mathcal{E}[\![\mathrm{E}]\!]\rho\sigma'\ \mathit{then}\ f(\mathcal{C}[\![\mathrm{C}]\!]\rho\sigma')\ \mathit{else}\ \sigma'$$

It was inductive definitions, like the one above, that forced language designers to confront the *mathematics* of iteration (in terms of least-fixed-point calculation).

3 Semantic-Domain Definitions

Perhaps Strachey's most important individual contribution to language engineering was his invention of semantic domains to name and organize into sets a language's semantic values. If denotational semantics is "BNF extended to semantics," then semantic-domain equations are "data typing extended to semantics." By using standard names (*expressible value, denotable value, storable value, environment, store, continuation, etc.*) for the semantic domains, Strachey created both a lexicon and a framework for describing a programming language's expressibility in advance of its syntax definitions and its valuation functions.

As Strachey noted himself in his paper, *The Varieties of Programming Language* [48], every language has *elementary*, *derived*, and *characteristic* domains. Elementary domains are sets of primitive values, like integers, strings, and storage locations; derived domains are standard structures, like environments and stores; and the characteristic domains are the ones that give the language its personality—for Strachey, these were the denotables (values that identifiers can denote), expressibles (values that expressions can denote) and the storables (values that can be stored). Figure 3 presents a comparison of the domains of simple Fortran-like and Lisp-like languages.

What we learn from reading the Figure is striking: The Fortran-like language uses a “flat” store, where only integers are saved. Only locations and statically scoped, parameterless procedures can be named, and expressions must evaluate to integers. The apparent application of the Fortran-like language is to numerical computing. In contrast, the Lisp-like language lacks primary storage but allows atoms, lists, and dynamically scoped functions to be computed; everything that can be computed can be named. The Lisp-like language is clearly suited for symbolic computation on linear data structures.

Strachey intended that a language designer begin work by listing the semantic-domain equations. Not only do the domains define a language's capabilities, they suggest a syntax (by means of the characteristic domains). There even exist “taxonomies” of languages—as suggested by Figure 3, a typical declarative language will have a rich collection of denotables but a paltry collection of storables, whereas an imperative language might have exactly the opposite balance.

Occasionally, one encounters a domain equation that is self referential:

$$List = Nil + (Atom \times List)$$

Fortunately, most such recursions can be resolved by the usual inductive interpretation that one applies to BNF definitions. The exception to this rule is an equation whose recursion lies to the left of a function-space constructor, e.g.,

$$\begin{aligned}&Function = Environment \rightarrow Expressible \rightarrow Expressible\\ &Expressible = Atom + List + Function\end{aligned}$$

Although the discovery of a set that satisfies such a definition to isomorphism was a significant achievement [42, 45], the attention surrounding this particular problem generated the unfortunate impression that denotational semantics was a topic better left to mathematicians rather than to language engineers! Of course, this is not the case—semantic domains are a powerful engineering tool and its users need not understand its technicalities any more deeply than a practicing civil engineer needs to understand the foundations of functional analysis to solve a differential equation.

4 Calculi for Semantic Description

By using lambda-calculus notation—abstraction, application, and identifier reference—to define the semantics of individual programming constructs, Strachey showed that semantics could be both elegant and concise and that *a language construct can have a meaning independent of any particular machine.*

Here is an example. The semantics of sequential composition of commands is presented in lambda-calculus as the application of the first command to the store and the application of the second command to the result of the first:

$$\begin{aligned}
&\mathcal{C}[\![\mathrm{C}_1; \mathrm{C}_2]\!] = \lambda\rho.\lambda\sigma.\ (\mathcal{C}[\![\mathrm{C}_2]\!]\rho)(\mathcal{C}[\![\mathrm{C}_1]\!]\rho\sigma)\\
&\quad \text{where}\\
&\qquad \rho \in \mathit{Environment},\\
&\qquad \sigma \in \mathit{Store},\\
&\qquad \mathcal{C} : \text{Command} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow \mathit{Store}
\end{aligned}$$

The minimality of lambda-calculus makes the semantics simple and direct. Yet, the semantics reveals these deep insights:

- the usage of environment implies that command sequencing is statically scoped—bindings made by C_1 do not affect C_2.
- the usage of store implies that command sequencing is truly sequential—C_2 depends on C_1's updates
- because it is stated in terms of function application, command composition is associative

These facts are uncovered *without recourse to specific machine models*. Also, the nested application expressed in the semantics of command sequencing states a *pattern* that a careful language designer will repeat in the semantics of declaration composition, expression composition, and so on. Such terse but exactly correct depictions are a revelation when compared to their counterparts, namely, machine-based operational-semantics definitions.

Of course, lambda notation must itself have meaning to be useful. In this regard, the notation is particularly well chosen, because its crucial computational property is captured by a *single* reduction rule,

$$(\lambda x.e_1)e_2 \implies [e_2/x]e_1$$

With this one concept—the binding of a value to a name—Strachey built towers of meaning, encompassing storage creation and update, declaration processing, abstraction and encapsulation mechanisms, and data and control flow.

Scott justified the above reduction rule by proving it to be a mathematical equality within a mathematical semantics of lambda-calculus, where an abstraction expression, $\lambda x.e$, denotes a mathematical function. Scott also exposed that the mathematical semantics of a while-loop (see the equation at the end of Section 2) can be understood as the least fixed-point solution of the recursively defined function with regard to a partially ordered set of stores.

But again, a language engineer need not know all the underlying physics of lambda-calculus denotations to perform useful language design with it.

5 Achievements from Denotational Semantics

Because of Strachey's fundamental contributions, researchers were able to give semantic definitions to languages like Ada, Algol60, Algol68, Lisp, Pascal, Scheme, and Snobol [4, 14, 27, 32, 35, 49, 50]. But perhaps the most important consequence of Strachey's work was the stimulation of research on the theory of programming languages.

One prominent example was Tennent's astute observation that Landin's "correspondence principle" [25] regarding semantics of definition-by-declaration and definition-by-parameter-passing was in fact an example of a recurring pattern of semantics. As Landin stated, the semantics of a declaration of `I`:

```
let I = E1 in E2
```

should be "the same as" the semantics of parameter binding to `I`:

```
procedure I2(I) begin E2 end ...  I2(E1) ...
```

As Tennent noted, this correspondence is exposed as two uses of the same semantical notational pattern in the semantics of the two respective phrases [51].

Following this tack, Tennent proposed other situations where semantical patterns can be used to design a language with internal coherency in binding [52]:

- *abstraction principle*: the elements of any semantically meaningful syntactic class can be named, and the names can be invoked (e.g., a procedure is a named command, a module is a named declaration, a class is a named type-structure, etc.).
- *qualification principle*: every semantically meaningful syntactic class may admit a block construct to hold local definitions (e.g., a command block is a command with its own local definitions, an encapsulated module is a declaration with its own local definitions, a class with local state is a class with its own local definitions, etc.).
- *parameterization principle* [43]: any named phrase can receive parameters from any semantically meaningful syntactic class (e.g., procedures may receive command parameters or declaration parameters or type-structure parameters, etc.).

Tennent's principles defined a taxonomy for binding mechanisms—a significant achievement, given that the evolution of binding mechanisms was perhaps the most important issue in language design during the 1960's and 1970's! (See Gelernter and Jagannathan's text for an excellent historical survey [13].)

Aware of similar possibilities in other facets of language definition, Mosses and Watt developed *action semantics*, a semantics metalanguage where there exist prebuilt semantical patterns for value flow, binding flow, control flow, and communication flow [33, 34]. Perhaps action semantics constitutes the most comprehensive body of knowledge today about how programming languages are assembled from underlying semantical patterns.

Because of its success as a semantics metalanguage, the lambda-calculus was intensively studied as a programming language in its own right. This opened the door to a variety of languages that were *higher-order* e.g., ML and Hope

[5, 29, 30], and also *lazy*, e.g., Miranda, and Haskell [53, 54]. Another consequence was renewed interest in compiler-generation and rapid-prototyping systems, where a language's implementation is mechanically synthesized from its semantic definition (as written in lambda-calculus).

The presence of semantic domains in a semantics definition meant that one could elegantly define a language's *abstract interpretation* by replacing one semantic domain by another that is a homomorphic image of the original. This generated a significant series of results that generalized flow-analysis techniques for flowchart languages to static analyses for modern, modular, higher-order languages. [8, 9, 21, 37, 38].

6 A Third Wave of Semantics Research

Each of Strachey's fundamental contributions has generated its own stream of research; we call the confluence of these steams the "Third Wave." Each is considered below.

6.1 Structure in Operational Semantics

Inductive language definition was, for many, the strongest selling point of denotational semantics, and a stream of research on inductively defined semantics followed. Kahn's *natural semantics* [22] dispensed with the mathematical aspects of denotational semantics (especially, the use of functions-as-values) and based its expressive power on *inductive derivability*, as expressed as logical-style inference rules. Here is the natural-semantics definition of command composition:

$$\frac{\sigma \vdash_\rho \mathrm{C}_1 \Downarrow \sigma_1 \qquad \sigma_1 \vdash_\rho \mathrm{C}_1 \Downarrow \sigma_2}{\sigma \vdash_\rho \mathrm{C}_1; \mathrm{C}_2 \Downarrow \sigma_2}$$

A program's semantics is calculated by proof-theoretic derivation of a sequent, $\sigma \vdash_\rho \mathrm{C} \Downarrow \sigma'$, which asserts that a starting store, σ, is used by a command, C, within an environment, ρ, to produce a revised store, σ'. Both σ and ρ are understood as first-order (tabular) data structures, replacing intellectual novelty by transparency.

Likewise, Plotkin proposed *structural operational semantics* as an inductively defined variant of a term-rewriting system, where the next redex to be contracted is selected by inductive derivability [16, 40]. Command composition is expressed as follows:

$$\frac{\mathrm{C}_1, \sigma \rightarrow_\rho \mathrm{C}_1', \sigma'}{\mathrm{C}_1; \mathrm{C}_2, \sigma \rightarrow_\rho \mathrm{C}_1'; \mathrm{C}_2, \sigma'} \qquad \frac{\mathrm{C}_1, \sigma \rightarrow_\rho \sigma'}{\mathrm{C}_1; \mathrm{C}_2, \sigma \rightarrow_\rho \mathrm{C}_2, \sigma'}$$

Plotkin's formulation triggered intensive interest in proving properties of programs that employ parallelism, because parallelism can be encoded in terms of interleavings with rules that resemble the two above.

Introduction rules:

$$\Gamma \vdash nil \in \phi List \qquad \frac{\Gamma \vdash e \in \phi \quad \Gamma \vdash e' \in \phi List}{\Gamma \vdash e :: e' \in \phi List}$$

Elimination rule:

$$\frac{\Gamma \vdash e \in \phi List \quad \Gamma \vdash e' \in \phi'_{nil} \quad \Gamma, a \in \phi, b \in \phi List, x \in \phi'_b \vdash e''_{a,b,x} \in \phi'_{a::b}}{\Gamma \vdash \mathit{cases}\ e\ \mathit{of}\ \mathit{is}(nil) : e' \,[]\, \mathit{is}(a :: b) : \mathit{assume}\ x \in \phi'_b\ \mathit{in}\ e''_{a,b,x} \in \phi'_e}$$

Computation rules:

$$\mathit{cases}\ nil\ \mathit{of}\ \mathit{is}(nil) : e' \,[]\, \mathit{is}(a :: b) : \mathit{assume}\ x \in \phi'_b\ \mathit{in}\ e''_{a,b,x} \implies e'$$

$$\begin{array}{l} \mathit{cases}\ (m :: \ell)\ \mathit{of}\ \mathit{is}(nil) : e' \,[]\, \mathit{is}(a :: b) : \mathit{assume}\ x \in \phi'_b\ \mathit{in}\ e''_{a,b,x} \\ \quad \implies \mathit{let}\ c = (\mathit{cases}\ \ell\ \mathit{of}\ \mathit{is}(nil) : e' \,[]\, \mathit{is}(a :: b) : \mathit{assume}\ x \in \phi'_b\ \mathit{in}\ e''_{a,b,x}) \\ \qquad \mathit{in}\ e''_{m,\ell,c} \end{array}$$

Figure 4: ITT definition of type $\phi List$

6.2 Type Theory

The substantial benefits provided by semantic domains led to investigations of enrichments. A stream of research on *type theory* was a consequence.

Strachey's domain calculus was a positive propositional calculus, where $+$ can be read as the logical connective $\vee$, $\times$ can be read as $\wedge$, and $\rightarrow$ can be read as $\supset$. Type theory added universal and existential quantification to the basic calculus: a type, $\forall x{:}\,\phi'.\phi_x$, named the collection of functions (lambda abstractions), f, such that for $e \in \phi'$, $f(e) \in \phi_e$.[4] Similarly, the existentially quantified type, $\exists x{:}\,\phi'.\phi_x$, named pairs, $\langle w, e_w \rangle$ such that $w \in \phi'$ and $e_w \in \phi_w$.[5] A further extension is *impredicative type theory*, where ϕ' is replaced by *Type*, the "type" of all data types.

Type theories with quantifiers are best used to define semantics of languages with parametric polymorphic functions and encapsulation mechanisms [7, 31, 41].

Perhaps the most striking of all type theories was Martin-Löf's *intuitionistic type theory* [39, 44], because new types are defined on demand in terms of natural-deduction-style introduction and elimination rules. The rules are accompanied by computation rules, which describe how computation proceeds on elements of the type.

Figure 4 gives the ITT formulation of a type, $\phi List$, of cons-lists. The figure is striking because the introduction rules define the semantical elements of the type inductively—a cons-list is either a nil structure or a cons-structure, where the latter contains a ϕ-element and a cons-structure. The introduction rules define the type's *canonical elements*. A type's elimination rule displays the

[4] $\forall x{:}\,\phi'.\phi_x$ is also written as $\Pi x : \phi'.\phi_x$.

[5] Again, $\exists x{:}\,\phi'.\phi_x$ is also written as $\Sigma x : \phi'.\phi_x$.

pattern for disassembly of canonical elements—a cons-list is disassembled by a cases analysis, where a nil-structure is processed directly and a cons-structure is processed by primitive recursion on its substructure. Finally, the computation rules show how an elimination rule "cancels" an introduction rule and generates primitive-recursive computation.

The heavily subscripted data types can be used to state correctness properties. For example, we might wish to write a function, *length*, that returns the length of its cons-list argument. We begin by asserting that $0 \in \mathit{lengthOf}_{nil}$ (literally, "0 is in the set of correct lengths of nil"), which suggests that $length(nil)$ should return 0 as its answer. Next, for a nonempty list, $a :: b$, we might assume that $length(b)$ returns lth, where $lth \in \mathit{lengthOf}_b$, and we use this assumption to conclude that $plus(1, lth) \in \mathit{lengthOf}_{a::b}$. Hence, $length(a :: b)$ should return $plus(1, lth)$.

In ITT, *a function's coding is its proof of correctness, because the correctness property is stated as the function's data type*:

$$
\begin{array}{rl}
\mathit{length}(\ell) = & \mathit{cases}\ \ \ell\ \mathit{of} \\
& \qquad \mathit{is}(nil) : 0 \\
& \quad\ [\,]\ \mathit{is}(a :: b) : \mathit{assume}\ \ \mathit{lth} \in \mathit{lengthOf}_b \\
& \qquad\qquad\qquad\quad \mathit{in}\ \ plus(1, \mathit{lth}) \\
& \in \mathit{lengthOf}_\ell
\end{array}
$$

The elimination rule in Figure 4 is used to type check the function and validate its typing, namely, $length(\ell) \in \mathit{lengthOf}_\ell$, for all $\ell \in \phi List$.

ITT provides a domain-driven, proof-theoretic definition of computation. Indeed, the syntax and computation rule for the lambda-calculus is in fact derived from the ITT formulation of the logical implication operator, $\supset$.

6.3 Semantic Calculi

The lambda-calculus expressed one concept well—the binding of value to name—and it is remarkable that all of computation can be encoded in this style. The cost one pays, however, is that control-flow and data-structuring concepts often have tedious codings in the calculus. Indeed, if a particular semantic feature must be intensively studied, it might be better to design a calculus that expresses well precisely that feature.

Building on denotational semantics' successes with lambda-calculus, other calculi have been developed for intensive analysis of particular semantic features. Examples of such calculi are

- *Calculus for communicating systems* [28]: a calculus for study of process communication. A basic computation step takes the form,

  $$\bar{\alpha}v.p \mid\mid \alpha x.q_x \overset{\alpha v}{\to} p \mid\mid q_v$$

  where one process sends a message, v, to another along a "channel," α.

- *Object calculus* [1]: a calculus for study of object-based message passing. A computation step looks like this:

$$o.l_j \rightarrow [o/\mathit{self}]b_j, \qquad \text{where } o = \{l_i = \varsigma(\mathit{self}).b_i\}_{i \in 1..n}$$

  where an object, o, with methods labelled l_i, $i \in 1..n$, receives a message to execute its method, l_j. *Self* references within l_j's body are dynamically resolved to o itself.

- *Ambient calculus* [6]: a calculus for study of agent transfer on the Internet.

- *Security-π and SLam calculi* [2, 17]: calculi for study of secure communications.

Just as the lambda-calculus opened the door to intensive study of binding, data typings for binding, and taxonomies of binding, the specialized calculi listed above have initiated similar lines of research into the specific language features for which they are designed. Indeed, each calculus suggests a *paradigm* of computation, and a language might be classified by paradigm in terms of which calculus provides for it the most elegant semantics. Initial studies of *multi-paradigm languages* [26] have been made in terms of unioning two or more such specialized calculi.

7 Impact on the Mainstream

The three innovations led by Strachey—inductive definition of semantics, semantic-domain definitions, and semantic calculi—are slowly but clearly moving into the programming mainstream. For example, a reader of *The Java Language Specification* [15], the reference manual for the core Java language, cannot help but notice the book's organization in terms of inductive definition. Semantic domains are implicitly used, often couched in the terminology of data structures, but the domains exist, nonetheless.

A more overt use of Strachey's work appears in *The Definition of Standard ML* [29, 30], where inductive definition, domains, and calculi are the primary tools for exposition of Standard ML.

Strachey's principles will become even more crucial as "domain specific" and "rapidly prototyped" special-purpose languages move to the center of the programming mainstream. A domain-specific language is built from a language core that is extended with operations for a specialized application area. Semantic domain definitions are crucial for defining the data and operations of the applications area, and an inductive definition is the key to integrating the language core with its application-specific extensions.

Yet another fundamental use of Strachey's principles is found in object-based *design patterns* [11], which are best understood as a collection of semantical notational patterns that one uses to assemble a programming language—design patterns form a "calculus" for object-based program construction.

Finally, it is difficult to imagine how the potentially unifying theory of *software architecture* [12, 47], that is, the study of large-scale structure of computer programs, would be undertaken without the groundwork that Strachey laid. Vestiges of inductive definition, domains, and semantic calculi appear throughout the architectural specifications and diagrams one writes to document a program's architecture.

Indeed, Christopher Strachey's ultimate achievement will most likely be found, not in the use of a particular notation or design principle, but in the subtle ways that his proposals lay out the paths that are taken by the generations of innovators who follow him. These paths are being explored now.

Acknowledgements

I would like to thank Neil Jones, who taught me denotational semantics; Peter Mosses, who answered many of my questions; and Olivier Danvy, who encouraged me to write this paper. I also thank Olivier, Andrzej Filinski and Stefan Sokolowski, who read earlier drafts of the paper.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, Berlin, 1996.

[2] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the SPi calculus. *Information and Computation*, 148:1–70, 1999.

[3] D.W. Barron. *An Introduction to the Study of Programming Languages.* Cambridge Univ. Press, 1977.

[4] D. Bjørner and O. Oest. *Towards a formal description of Ada.* Lecture Notes in Computer Science 98. Springer Verlag, 1980.

[5] R. Burstall, D. MacQueen, and D. Sanella. HOPE: an experimental applicative language. Technical Report CSR-62-80, Computer Science Dept., University of Edinburgh, 1981.

[6] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming*, Lecture Notes in Computer Science 1603, pages 51–94. Springer Verlag, 1999.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[9] P. Cousot and R. Cousot. Higher-order abstract interpretation. In *Proc. IEEE Int'l. Conf. Programming Languages.* IEEE Press, 1994.

[10] R.W. Floyd. Assigning meanings to programs. *Symposia in Applied Mathematics*, 19:19–32, 1967.

[11] E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[12] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge*, pages 1–39. World Scientific, 1993.

[13] D. Gelernter and S. Jagannathan. *Programming Linguistics*. MIT Press, Cambridge, MA, 1990.

[14] M.J.C. Gordon. Towards a semantic theory of dynamic binding. Technical Report STAN-CS-75-507, Computer Science Dept., Stanford Univ., 1975.

[15] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, MA, 1996.

[16] C. Gunter. *Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1992.

[17] N. Heintze and J. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, 1998.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[19] C.A.R. Hoare. Notes on data structuring. In O.J. Dahl, E. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press, 1972.

[20] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[21] N. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4*, pages 527–636. Oxford Univ. Press, 1995.

[22] G. Kahn. Natural semantics. In *Proc. STACS '87*, pages 22–39. Lecture Notes in Computer Science 247, Springer, Berlin, 1987.

[23] D. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[24] D. Knuth. The semantics of context-free languages. *Math. Systems Theory*, 2:127–145, 1971. Corrigenda, Vol 5., p.95, 1971.

[25] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[26] J. Lloyd. Combining functional and logic programming languages. In M. Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 Symposium*, pages 43–57. MIT Press, 1994.

[27] R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[28] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[29] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[30] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[31] J.C. Mitchell. *Foundations for Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1996.

[32] P. D. Mosses. Mathematical semantics of Algol60. Technical Report Technical Monograph PRG-12, Programming Research Group, University of Oxford, U.K., 1974.

[33] P. D. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[34] P.D. Mosses and D.A. Watt. The use of action semantics. In Martin Wirsing, editor, *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986.* IFIP, North-Holland, 1987.

[35] S. Muchnick and U. Pleban. A semantic comparison of LISP and Scheme. In *Conf. on LISP and Functional Programming*, pages 56–64. ACM, 1982.

[36] P. Naur. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[37] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[38] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[39] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* International Series on Monographs on Computer Science No. 7. Oxford University Press, 1990.

[40] G.D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[41] J. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[42] J.C. Reynolds. Notes on a lattice-theoretic approach to the theory of computation. Technical report, Computer Science, Syracuse University, 1972.

[43] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[44] D.A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, 1994.

[45] D.S. Scott. Outline of a mathematical theory of computation. Technical Report Report PRG-2, Prog. Research Group, Oxford University, 1971.

[46] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of Symposium on Computers and Automata*, pages 19–46. Microwave Research Institute Symposia Series: Volume 21, Polytechnic Institute of Brooklyn, 1971.

[47] M. Shaw and D. Garlan. *Software Architecture: Pespectives on an Emerging Discipline*. Prentice Hall, 1996.

[48] C. Strachey. The varieties of programming language. Technical Report PRG-10, Prog. Research Group, Oxford University, 1973.

[49] R.D. Tennent. Mathematical semantics of SNOBOL4. In *POPL'73, Conference Record of the 1st Annual ACM Symposium on Principles of Programming Languages*, pages 95–107, 1973.

[50] R.D. Tennent. A denotational description of the programming language Pascal. Technical Report 77-47, Dept. of Computing and Information Sciences, Queen's University, Kingston, Ontario, Canada, 1977.

[51] R.D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[52] R.D. Tennent. *Principles of Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

[53] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Reading, MA, 1999.

[54] D. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21:158–166, 1986.