The Essence of Reynolds

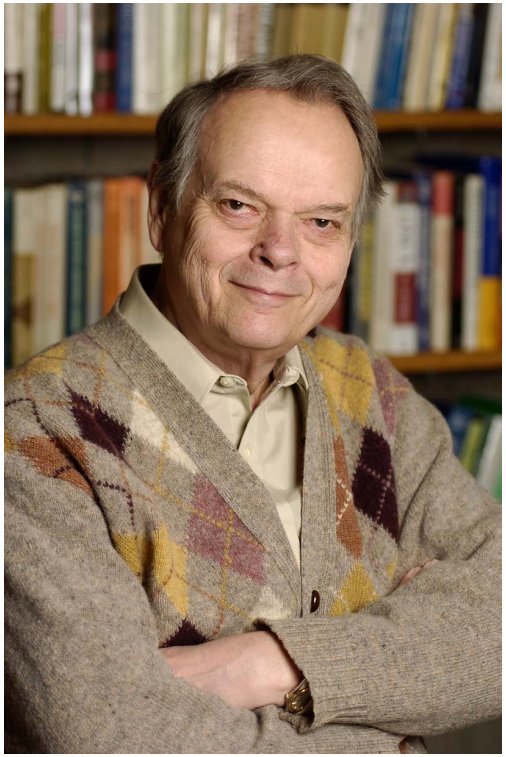
(title borrowed from POPL14's session
in honour of John C. Reynolds,
by Stephen Brookes, Peter O'Hearn and Uday Reddy)

Álvaro García-Pérez

Reykjavík University

April 10th, 2015

John C. Reynolds (1935-2013)



(1953-56) Purdue University. B.S.

(1956-61) Harvard University. Ph.D.

(1961-70) Argonne National Laboratory.

Part-time Professorial Lecturer at University of Chicago and Assistant Professor at Stanford University.

(1970-86) Syracuse University. Professor of Computer Science.

(1986-2013) Carnegie Mellon University. Professor of Computer Science.

Most research fields are apple trees: most of the low-hanging fruit have already been picked, so you have to be a good climber to get the leftovers. The great researchers are the ones that can spot new trees.—Peter O'Hearn

Ph.D. Thesis

(1961) *Surface Properties of Nuclear Matter*

Ph.D. Thesis

(1961) *Surface Properties of Nuclear Matter*

> *A big number-crunching program designed to produce an uninteresting computation of an unimportant quantity in a bad approxiamtion.*—John Reynolds

Early works (1961-69)

(1965) *COGENT, A Compiler and Generalized Translator*

- Early compiler generator.

Early works (1961-69)

(1965) *COGENT, A Compiler and Generalized Translator*

- Early compiler generator.

(1968) *Automatic Computation of Data Set Definitions*

- Early abstract interpretation, more than a decade ahead of time!

Early works (1961-69)

(1965) *COGENT, A Compiler and Generalized Translator*

- Early compiler generator.

(1968) *Automatic Computation of Data Set Definitions*

- Early abstract interpretation, more than a decade ahead of time!

(1969) *Transformational Systems and the Algebraic Structure of Atomic Formulas*

- Anti-unification algorithm. Basis of inductive logic programming.

Early works (1961-69)

(1965) *COGENT, A Compiler and Generalized Translator*

- Early compiler generator.

(196

(196

> *These works [. . .] would already have given a very respectable career. But it was now that John entered a kind of golden period, and extended run of high productivity and extreme quality.*—Brookes, O'Hearn, Reddy

Golden Period (1970-84)

(1970) *GEDANKEN - a Symple Typeless Language Based on the Principle of Completeness and the Reference Concept*

- Untyped call-by-value language with references.

GEDANKEN–A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept

John C. Reynolds
Argonne National Laboratory, Argonne, Illinois*

GEDANKEN is an experimental programming language with the following characteristics. (1) Any value which is permitted in some context of the language is permissible in any other meaningful context. In particular, functions and labels are permissible results of functions and values of variables. (2) Assignment and indirect addressing are formalized by introducing values, called references, which in turn possess other values. The assignment operation always affects the relation between some reference and its value. (3) All compound data structures are treated as functions. (4) Type declarations are not permitted.

The functional approach to data structures and the use of references insure that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be implicit; i.e. it may be specified by giving an arbitrary algorithm for computing or accessing its components. The existence of label variables permits the construction of coroutines, quasi-parallel processes, and other unorthodox control

context of the language is permissible in any other meaningful context. In particular, functions and labels are permitted to be results of functions or values of references (e.g. variables), without imposing restrictions which maintain a stack discipline for run-time storage allocation.

(2) *The Reference Concept.* Assignment and indirect addressing are formalized in the following manner: among the possible values which may occur in a program are objects called *references*, which in turn possess other values. The assignment operation always affects the relation between some reference and its value.

Neither of these principles is novel. Lisp [1a and 1b] (in its interpretive implementations), Iswim [2], and Pal [3] all satisfy the principle of completeness, and the reference concept is used in Algol 68 [4] and Basel [5]. But Gedanken goes beyond these languages in exploiting the power of these principles, i.e. in eliminating other language features which are rendered redundant by completeness and references. Specifically:

(1) The existence of function-returning and reference-returning functions allows all compound data structures to be treated as functions. For example, a one-dimensional Algol-like array is treated as a function whose domain is a finite set of consecutive integers and which maps each of these integers into a unique reference. This approach insures that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be *implicit*; i.e. it may be specified by giving an arbitrary algorithm for computing or accessing its compo-

(1970) *GEDANKEN - a Symple Typeless Language Based on the Principle of Completeness and the Reference Concept*

- Untyped call-by-value language with references.

GEDANKEN–A Simple Typeless Language Based on the Principle of Completeness and the

I'll never design another untyped language.—John Reynolds

GEDANKEN is an experimental programming language with the following characteristics. (1) Any value which is permitted in some context of the language is permissible in any other meaningful context. In particular, functions and labels are permissible results of functions and values of variables. (2) Assignment and indirect addressing are formalized by introducing values, called references, which in turn possess other values. The assignment operation always affects the relation between some reference and its value. (3) All compound data structures are treated as functions. (4) Type declarations are not permitted.

The functional approach to data structures and the use of references insure that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be implicit; i.e. it may be specified by giving an arbitrary algorithm for computing or accessing its components. The existence of label variables permits the construction of coroutines, quasi-parallel processes, and other unorthodox control

context of the language is permissible in any other meaningful context. In particular, functions and labels are permitted to be results of functions or values of references (e.g. variables), without imposing restrictions which maintain a stack discipline for run-time storage allocation.

twech some reference and its value.

Neither of these principles is novel. LISP [1a and 1b] (in its interpretive implementations), ISWIM [2], and PAL [3] all satisfy the principle of completeness, and the reference concept is used in ALGOL 68 [4] and BASEL [5]. But GEDANKEN goes beyond these languages in exploiting the power of these principles, i.e. in eliminating other language features which are rendered redundant by completeness and references. Specifically:

(1) The existence of function-returning and reference-returning functions allows all compound data structures to be treated as functions. For example, a one-dimensional ALGOL-like array is treated as a function whose domain is a finite set of consecutive integers and which maps each of these integers into a unique reference. This approach insures that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be *implicit*; i.e. it may be specified by giving an arbitrary algorithm for computing or accessing its components. (Functional data structures have been suggested by

(1972) *Definitional Intepreters for Higher-Order Programming langugages*

- Introduces defunctionalization.

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call-by-value versus call-by-name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in

INTRODUCTION

An important and frequently used method of defining a programming language is to give an interpreter for the language which is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have tried to make the presentation accessible to readers who are unfamiliar with this previous work.

(Of course, interpretation can provide an implementation as well as a definition, but there are large practical

(1972) *Definitional Intepreters for Higher-Order Programming langugages*

- Introduces defunctionalization.

```
eval = λ(r, e).
    (const?(r) → evcon(r),
    var?(r) → e(r),
    appl?(r) → (eval(opr(r), e))(eval(opnd(r),e)),
    lambda?(r) → evlambda(r, e),
    cond?(r) → if eval(prem(r), e)
        then eval(conc(r), e) else eval(altr(r),e),
    letrec?(r) → letrec e' =
        λx. if x = dvar(r) then evlambda(dexp(r), e') else e(x)
        ------------------------------------------------------
        in eval(body(r), e'))
evlambda = λ(ℓ, e). λa. eval(body(ℓ), ext(fp(ℓ), a, e))
ext = λ(z, a, e). λx. if x = z then a else e(x)
                      ---------------------------
```

C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call-by-value versus call-by-name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in

derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have tried to make the presentation accessible to readers who are unfamiliar with this previous work.

(Of course, interpretation can provide an implementation as well as a definition, but there are large practical

(1972) *Definitional Intepreters for Higher-Order Programming langugages*

- Introduces defunctionalization.

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Olivier Danvy—*So, in your opinion, what would be the intended semantics of the meta-language?*

procedures or labels
) are usually defined
ch are themselves
mming language based
lus (i.e., an
e such as pure LISP).
Carthy's definition
ECD machine, the
f PL/I, Reynolds'
NKEN, and recent
L. Morris and
definitions can be
g to whether the
s higher-order
her the order of
call-by-value versus
call-by-name) in the defined language
depends upon the order of application
in the defining language. As an example,
we consider the definition of a simple
applicative programming language by
means of an interpreter written in
a similar language. Definitions in

An important and frequently used
method of defining a programming language
is to give an interpreter for the language
which is written in a second, hopefully
better understood language. (We will
call these two languages the *defined*
and *defining* languages, respectively.)
In this paper, we will describe and
classify several varieties of such
interpreters, and show how they may be
derived from one another by informal but
constructive methods. Although our
approach to "constructive classification"
is original, the paper is basically an
attempt to review and systematize
previous work in the field, and we have
tried to make the presentation accessible
to readers who are unfamiliar with this
previous work.

(Of course, interpretation can
provide an implementation as well as a
definition, but there are large practical

(1972) *Definitional Intepreters for Higher-Order Programming langugages*

- Introduces defunctionalization.

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Olivier Danvy—*So, in your opinion, what would be the intended semantics of the meta-language?*
Peter Landin—*Call-by-value, of course!*

procedures or l
) are usually d
ch are themselv
mming language
lus (i.e., an
e such as pure
Carthy's defini
ECD machine, th
f PL/I, Reynold
NKEN, and recen
L. Morris and
definitions ca
g to whether th
s higher-order
her the order o
call-by-value v
call-by-name) in the defined language
depends upon the order of application
in the defining language. As an example,
we consider the definition of a simple
applicative programming language by
means of an interpreter written in
a similar language. Definitions in

portant and frequently used
f defining a programming language
ve an interpreter for the language
written in a second, hopefully
nderstood language. (We will
se two languages the *defined*
ning languages, respectively.)
paper, we will describe and
several varieties of such
ters, and show how they may be
from one another by informal but
tive methods. Although our
to "constructive classification"
nal, the paper is basically an
to review and systematize
previous work in the field, and we have
tried to make the presentation accessible
to readers who are unfamiliar with this
previous work.
(Of course, interpretation can
provide an implementation as well as a
definition, but there are large practical

(1972) *Definitional Intepreters for Higher-Order Programming langugages*

- Introduces defunctionalization.

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Olivier Danvy—*So, in your opinion, what would be the intended semantics of the meta-language?*
Peter Landin—*Call-by-value, of course!*
John Reynolds—*Call-by-name, of course!*

procedures or l
) are usually d
ch are themselv
mming language
lus (i.e., an
e such as pure
Carthy's defini
ECD machine, th
f PL/I, Reynold
NKEN, and recen
L. Morris and
definitions ca
g to whether th
s higher-order
her the order o
call-by-value versus
call-by-name) in the defined language
depends upon the order of application
in the defining language. As an example,
we consider the definition of a simple
applicative programming language by
means of an interpreter written in
a similar language. Definitions in

portant and fre
f defining a pr
ve an interpret
written in a s
nderstood langu
se two language
ning languages,
paper, we will
several variet
ters, and show
from one anothe
tive methods.
to "constructi
nal, the paper
attempt to review and s
previous work in the field, and we have
tried to make the presentation accessible
to readers who are unfamiliar with this
previous work.
(Of course, interpretation can
provide an implementation as well as a
definition, but there are large practical

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

This is a preprint of a paper to be given at the Colloquium on Programming, Paris, 9-11 April 1974.

TOWARDS A THEORY OF TYPE STRUCTURE †

John C. Reynolds

Syracuse University

Syracuse, New York 13210, U.S.A.

Introduction

The type structure of programming languages has been the subject of an active development characterized by continued controversy over basic principles.(1-7) In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

$$
\begin{array}{lcl}
t & ::= & \underline{\texttt{integer}} \mid \underline{\texttt{real}} \\
 & \mid & t \to t \\
e & ::= & c \\
 & \mid & x \\
 & \mid & \lambda x \,\varepsilon\, t.\ e \\
 & \mid & e\ e
\end{array}
$$

paper to be given at the Colloquium on Programming,

ARDS A THEORY OF TYPE STRUCTURE †

John C. Reynolds

Syracuse University

racuse, New York 13210, U.S.A.

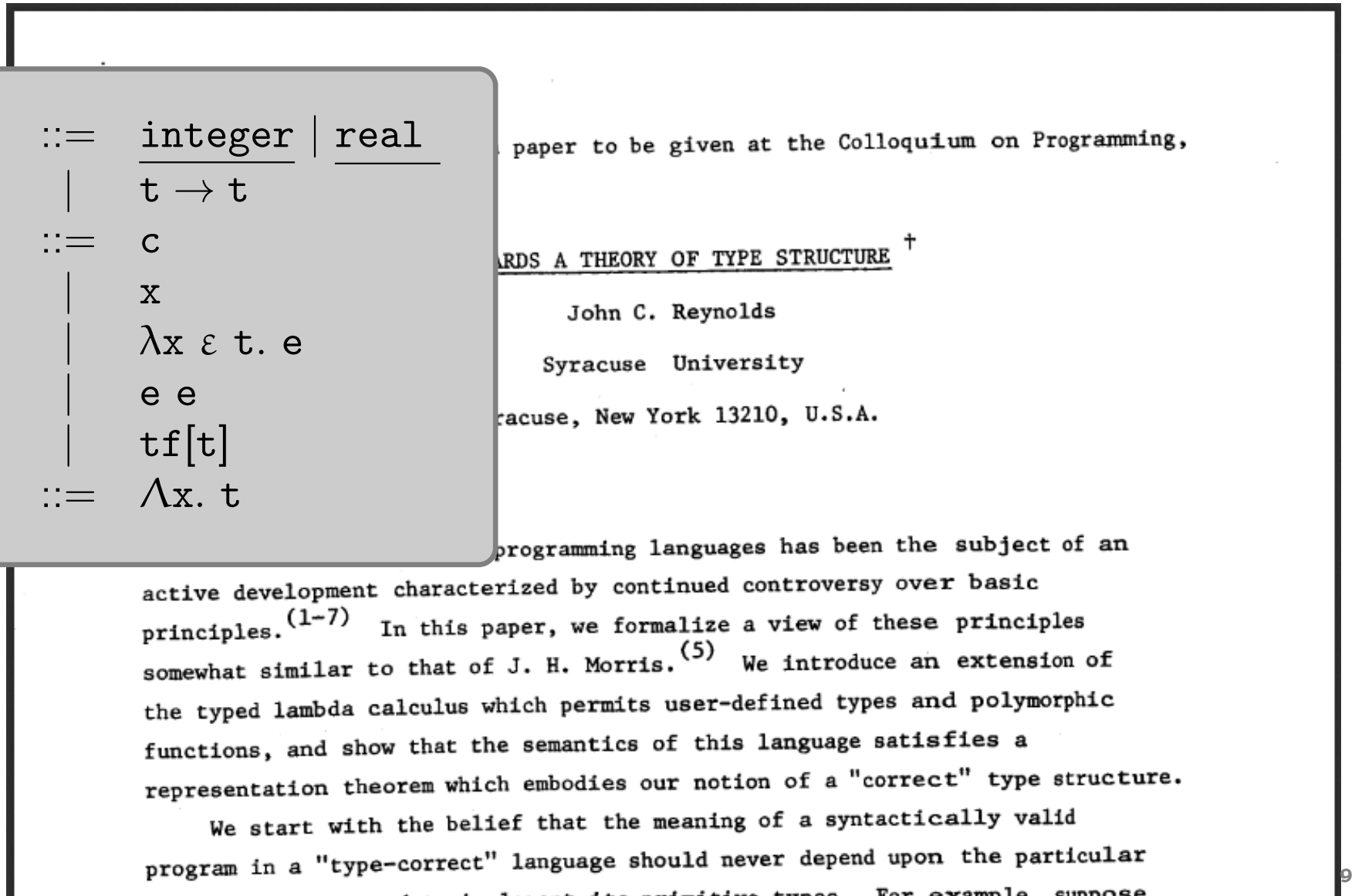
programming languages has been the subject of an active development characterized by continued controversy over basic principles.(1-7) In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

$$
\begin{array}{rcl}
\mathtt{t} & ::= & \underline{\mathtt{integer}} \mid \underline{\mathtt{real}} \\
 & \mid & \mathtt{t} \to \mathtt{t} \\
\mathtt{e} & ::= & \mathtt{c} \\
 & \mid & \mathtt{x} \\
 & \mid & \lambda \mathtt{x}\ \varepsilon\ \mathtt{t}.\ \mathtt{e} \\
 & \mid & \mathtt{e}\ \mathtt{e} \\
 & \mid & \mathtt{tf}[\mathtt{t}] \\
\mathtt{tf} & ::= & \Lambda \mathtt{x}.\ \mathtt{t}
\end{array}
$$



paper to be given at the Colloquium on Programming,

ARDS A THEORY OF TYPE STRUCTURE †

John C. Reynolds

Syracuse University

racuse, New York 13210, U.S.A.

programming languages has been the subject of an active development characterized by continued controversy over basic principles.(1-7) In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

$$\Lambda t.\ \lambda x\ \varepsilon\ t.\ x$$

$$
\begin{array}{rcl}
t & ::= & \underline{\texttt{integer}} \mid \underline{\texttt{real}} \\
 & \mid & t \to t \\
e & ::= & c \\
 & \mid & x \\
 & \mid & \lambda x\ \varepsilon\ t.\ e \\
 & \mid & e\ e \\
 & \mid & tf[t] \\
tf & ::= & \Lambda x.\ t
\end{array}
$$

on Programming,

ARDS A THEORY OF TYPE STRUCTURE †

John C. Reynolds

Syracuse University

racuse, New York 13210, U.S.A.

programming languages has been the subject of an active development characterized by continued controversy over basic principles.(1-7) In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

$$\Lambda t.\ \lambda x\ \varepsilon\ t.\ x$$

$$(\Lambda t.\ \lambda x\ \varepsilon\ t.\ x)[\underline{\texttt{integer}} \rightarrow \underline{\texttt{real}}]$$

$$\begin{array}{rcl} t & ::= & \underline{\texttt{integer}} \mid \underline{\texttt{real}} \\ & \mid & t \rightarrow t \\ e & ::= & c \\ & \mid & x \\ & \mid & \lambda x\ \varepsilon\ t.\ e \\ & \mid & e\ e \\ & \mid & tf[t] \\ tf & ::= & \Lambda x.\ t \end{array}$$

on Programming,

racuse, New York 13210, U.S.A.

programming languages has been the subject of an active development characterized by continued controversy over basic principles.[1-7] In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.[5] We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

$$
\begin{array}{rcl}
t & ::= & \underline{\texttt{integer}} \mid \underline{\texttt{real}} \\
 & \mid & t \to t \\
e & ::= & c \\
 & \mid & x \\
 & \mid & \lambda x \ \varepsilon \ t.\ e \\
 & \mid & e\ e \\
 & \mid & tf[t] \\
tf & ::= & \Lambda x.\ t
\end{array}
$$

$$\Lambda t.\ \lambda x\ \varepsilon\ t.\ x$$

$$(\Lambda t.\ \lambda x\ \varepsilon\ t.\ x)[\underline{\texttt{integer}} \to \underline{\texttt{real}}]$$

$$\lambda x\ \varepsilon\ \underline{\texttt{integer}} \to \underline{\texttt{real}}.\ x$$

on Programming,

racuse, New York 13210, U.S.A.

active development character
principles.(1-7) In this pap
somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1974) *Towards a Theory of Type Structure*

- Introduces the polymorphic lambda calculus ($\lambda 2$), which was discovered independently by Jean-Yves Girard.

This is a preprint of a paper to be given at the Colloquium on Programming, Paris, 9-11 April 1974.

We must admit a serious lacuna in our chain of argument.— John Reynolds

Syracuse, New York 13210, U.S.A.

Introduction

The type structure of programming languages has been the subject of an active development characterized by continued controversy over basic principles.(1-7) In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular

(1975) *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*

- Pinpointed the "expression problem", often revisited decades later.

This is a preprint of a paper to be given at the Conference on New Directions in Algorithmic Languages sponsored by IFIP Working Group 2.1, Munich, August 1975.

USER-DEFINED TYPES AND PROCEDURAL DATA STRUCTURES

AS COMPLEMENTARY APPROACHES TO DATA ABSTRACTION

John C. Reynolds

Syracuse University, Syracuse, New York

ABSTRACT User-defined types (or modes) and procedural (or functional) data structures are complementary methods for data abstraction, each providing a capability lacked by the other. With user-defined types, all information about the representation of a particular kind of data is centralized in a type definition and hidden from the rest of the program. With procedural data structures, each part of the program which creates data can specify its own representation, independently of any representations used elsewhere for the same

(1975) *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*

- Pinpointed the "expression problem", often revisited decades later.

This is a preprint of a paper to be given at the Conference on New Directions in Algorithmic Languages sponsored by IFIP Working Group 2.1, Munich, August 1975.

You too? I've long been trying to catch up to where I was.— John Reynolds

John C. Reynolds

Syracuse University, Syracuse, New York

ABSTRACT User-defined types (or modes) and procedural (or functional) data structures are complementary methods for data abstraction, each providing a capability lacked by the other. With user-defined types, all information about the representation of a particular kind of data is centralized in a type definition and hidden from the rest of the program. With procedural data structures, each part of the program which creates data can specify its own representation, independently of any representations used elsewhere for the same

(1981) *The Essence of Algol*

- Typed λ-calculus with subtypes, introduces semantics for local state.

The Essence of Algol[1]

John C. Reynolds
Syracuse University, Syracuse, NY, U.S.A.

Abstract

Although Algol 60 has been uniquely influential in programming language design, its descendants have been significantly different than their prototype. In this paper, we enumerate the principles that we believe embody the essence of Algol, describe a model that satisfies these principles, and illustrate this model with a language that, while more uniform and general, retains the character of Algol.

1. The Influence of Models of Algol

Among programming languages, Algol 60 [1] has been uniquely influential in the theory and practice of language design. It has inspired a variety of models which have in turn inspired a

(1981) *The Essence of Algol*

- Typed λ-calculus with subtypes, introduces semantics for local state.

What is less well known, and perhaps more striking, is the sheer number of absolutely first-rate contributions made by John across a broad range, all of which display his characteristic deep insight.—Brookes, O'Hearn, and Reddy

Abstract

Although Algol 60 has been uniquely influential in programming language design, its descendants have been significantly different than their prototype. In this paper, we enumerate the principles that we believe embody the essence of Algol, describe a model that satisfies these principles, and illustrate this model with a language that, while more uniform and general, retains the character of Algol.

1. The Influence of Models of Algol

Among programming languages, Algol 60 [1] has been uniquely influential in the theory and practice of language design. It has inspired a variety of models which have in turn inspired a

(1983) *Types, Abstraction and Parametric Polymorphism*

- Formalizes parametric polymorphism (Abstraction Theorem).

INFORMATION PROCESSING 83, R.E.A. Mason (ed.)
Elsevier Science Publishers B.V. (North-Holland)

TYPES, ABSTRACTION AND PARAMETRIC POLYMORPHISM†

John C. REYNOLDS
Syracuse University
Syracuse, New York, USA

Invited Paper

We explore the thesis that type structure is a syntactic discipline for maintaining levels of abstraction. Traditionally, this view has been formalized algebraically, but the algebraic approach fails to encompass higher-order functions. For this purpose, it is necessary to generalize homomorphic functions to relations; the result is an "abstraction" theorem that is applicable to the typed lambda calculus and various extensions, including user-defined types.

Finally, we consider polymorphic functions, and show that the abstraction theorem captures Strachey's concept of parametric, as opposed to ad hoc, polymorphism.

1. A FABLE

Once upon a time, there was a university with a peculiar tenure policy. All faculty were tenured, and could only be dismissed for moral turpitude. What was peculiar was the definition of moral turpitude: making a false statement in class. Needless to say, the university did not teach computer science. However, it had a renowned department of mathematics.

One semester, there was such a large enrollment in complex variables that two sections were scheduled. In one section, Professor Descartes announced that a complex number was an ordered pair of reals, and that two complex numbers were equal when their corresponding components were equal. He went on to explain how to convert reals into complex numbers, what "i"

The moral of this fable is that:

> Type structure is a syntactic discipline for enforcing levels of abstraction.

For instance, when Descartes introduced the complex plane, this discipline prevented him from saying Complex = Real × Real, which would have contradicted Bessel's definition. Instead, he defined the mapping f: Real × Real → Complex such that f(x, y) = x + i × y, and proved that this mapping is a bijection.

More subtly, although both lecturers introduced the set Int* of sequences of integers, and spoke of sets such as Int* → Complex, Int* × Complex, and Int* → Complex, they never mentioned Int* ∪ Complex or Int* ∩ Complex. Intuitively, they thought of sequences of integers and complex numbers as entities so

(1983) *Types, Abstraction and Parametric Polymorphism*

- Formalizes parametric polymorphism (Abstraction Theorem).

INFORMATION PROCESSING 83, R.E.A. Mason (ed.)
Elsevier Science Publishers B.V. (North-Holland)

So he locked himself in the library for six months and wrote "Types, Abstraction and Parametric Polymorphism". This is arguably the best paper on types ever written.—Peter O'Hearn

Once upon a time, there was a university with a peculiar tenure policy. All faculty were tenured, and could only be dismissed for moral turpitude. What was peculiar was the definition of moral turpitude: making a false statement in class. Needless to say, the university did not teach computer science. However, it had a renowned department of mathematics.

One semester, there was such a large enrollment in complex variables that two sections were scheduled. In one section, Professor Descartes announced that a complex number was an ordered pair of reals, and that two complex numbers were equal when their corresponding components were equal. He went on to explain how to convert reals into complex numbers, what "i"

Type structure is a syntactic discipline for enforcing levels of abstraction.

For instance, when Descartes introduced the complex plane, this discipline prevented him from saying Complex = Real × Real, which would have contradicted Bessel's definition. Instead, he defined the mapping f: Real × Real → Complex such that f(x, y) = x + i × y, and proved that this mapping is a bijection.

More subtly, although both lecturers introduced the set Int^* of sequences of integers, and spoke of sets such as Int^* + Complex, Int^* × Complex, and Int^* → Complex, they never mentioned Int^* ∪ Complex or Int^* ∩ Complex. Intuitively, they thought of sequences of integers and complex numbers as entities so

A fable (I/II)

Once upon a time, there was a university with a peculiar tenure policy. All faculty were tenured, and could only be dismissed for moral turpitude. What was peculiar was the definition of moral turpitude: making a false statement in class. Needless to say, the university did not teach computer science. However, it had a renowned department of mathematics.

One semester, there was such a large enrollment in complex variables that two sections were scheduled. In one section, Professor Descartes announced that a complex number was an ordered pair of reals, and that two complex numbers were equal when their corresponding components were equal. He went on to explain how to convert reals into complex numbers, what "i" was, how to add, multiply, and conjugate complex numbers, and how to find their magnitude.

A fable (II/II)

In the other section, Professor Bessel announced that a complex number was an ordered pair of reals the first of which was nonnegative, and that two complex numbers were equal if their first components were equal and either the first components were zero or the second components differed by a multiple of 2π. He then told an entirely different story about converting reals, "i", addition, multiplication, conjugation, and magnitude.

Then, after their first classes, an unfortunate mistake in the registrar's office caused the two sections to be interchanged. Despite this, neither Descartes nor Bessel ever committed moral turpitude, even though each was judged by the other's definitions. The reason was that they both had an intuitive understanding of type. Having defined complex numbers and the primitive operations upon them, thereafter they spoke at a level of abstraction that encompassed both of their definitions.

The moral of this fable is that:

> Type structure is a syntactic discipline for enforcing levels of abstraction.

Later works (1985-2013)

(1995) *Using Functor Category to Generate Intermediate Code*

- Connects category theory to compilation.

Using Functor Categories to Generate Intermediate Code *

John C. Reynolds

Department of Computing
Imperial College
London SW7 2BZ, Great Britain
jr@doc.ic.ac.uk

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
john.reynolds@cs.cmu.edu

Abstract

In the early 80's Oles and Reynolds devised a semantic model of Algol-like languages using a category of functors from a category of store shapes to the category of predomains. Here we will show how a variant of this idea can be used to define the translation of an Algol-like language to intermediate code in a uniform way that avoids unnecessary temporary variables, provides control-flow translation of boolean expressions, permits online expansion of procedures, and minimizes the storage overhead of calls of closed procedures. The basic idea is to replace continuations by instruction sequences and store shapes by descriptions of the structure of the run-time stack.

1 Introduction

To construct a compiler for a modern higher-level programming language, one needs to structure the translation to a machine-like intermediate language in a way that reflects the semantics of the language. Little is said about such structuring in compiler texts that are intended to cover a wide variety of programming languages. More is said in the literature on semantic-directed compiler construction [1]

2 Types and Syntax

An Algol-like language is a typed lambda calculus with an unusual repertoire of primitive types. Throughout most of this paper we assume that the primitive types are

$$\mathbf{comm}(\text{and}) \quad \mathbf{int}(\text{eger})\mathbf{exp}(\text{ression})$$

$$\mathbf{int}(\text{eger})\mathbf{acc}(\text{eptor}) \quad \mathbf{int}(\text{eger})\mathbf{var}(\text{iable}) ,$$

and that the set Θ of types is the least set containing these primitive types and closed under the binary operation $\rightarrow$.

We write $\leq$ for the least preorder such that

$$\mathbf{intvar} \leq \mathbf{intexp} \qquad \mathbf{intvar} \leq \mathbf{intacc}$$

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \rightarrow \theta_2 \leq \theta_1' \rightarrow \theta_2' .$$

When $\theta \leq \theta'$, θ is said to be a *subtype* of θ'.

A type assignment is a mapping from some finite set of identifiers into types; we write Θ^* for the set of type assignments. Then we write the *typing* $\pi \vdash p : \theta$ to indicate that the phrase p has type θ under the type assignment π.

We omit both the definition of the syntax of phrases and the inference rules for typings, beyond noting that phrases

(1995) *Using Functor Category to Generate Intermediate Code*

- Connects category theory to compilation.

Using Functor Categories
to Generate Intermediate Code *

John C. Reynolds

Programming languages semanticists should be the obstetricians of programming languages, not their coroners.—John Reynolds

In the early 80's Oles and Reynolds devised a semantic model of Algol-like languages using a category of functors from a category of store shapes to the category of predomains. Here we will show how a variant of this idea can be used to define the translation of an Algol-like language to intermediate code in a uniform way that avoids unnecessary temporary variables, provides control-flow translation of boolean expressions, permits online expansion of procedures, and minimizes the storage overhead of calls of closed procedures. The basic idea is to replace continuations by instruction sequences and store shapes by descriptions of the structure of the run-time stack.

1 Introduction

To construct a compiler for a modern higher-level programming language, one needs to structure the translation to a machine-like intermediate language in a way that reflects the semantics of the language. Little is said about such structuring in compiler texts that are intended to cover a wide variety of programming languages. More is said in the

An Algol-like language is a typed lambda calculus with an unusual repertoire of primitive types. Throughout most of this paper we assume that the primitive types are

$$\mathbf{comm}(\text{and}) \quad \mathbf{int}(\text{eger})\mathbf{exp}(\text{ression})$$

$$\mathbf{int}(\text{eger})\mathbf{acc}(\text{eptor}) \quad \mathbf{int}(\text{eger})\mathbf{var}(\text{iable}),$$

and that the set Θ of types is the least set containing these primitive types and closed under the binary operation $\rightarrow$.

We write $\leq$ for the least preorder such that

$$\mathbf{intvar} \leq \mathbf{intexp} \qquad \mathbf{intvar} \leq \mathbf{intacc}$$

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \rightarrow \theta_2 \leq \theta_1' \rightarrow \theta_2'.$$

When $\theta \leq \theta'$, θ is said to be a *subtype* of θ'.

A type assignment is a mapping from some finite set of identifiers into types; we write Θ^* for the set of type assignments. Then we write the *typing* $\pi \vdash p : \theta$ to indicate that the phrase p has type θ under the type assignment π.

We omit both the definition of the syntax of phrases and the inference rules for typings, beyond noting that phrases

(2000) *Intuitionistic Reasoning about Shared Mutable Data Structures*

- Not the final version of separation logic, but introduces the separating conjunction.

This is a revised draft of a paper to appear in "Millennial Perspectives in Computer Science", the proceedings of the Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, (held September 13–15, 1999), to be published by Palgrave. It supercedes the version (dated August 12, 1999) that was distributed at the meeting, which contained a serious error.

Intuitionistic Reasoning about Shared Mutable Data Structure*

John C. Reynolds
Department of Computer Science
Carnegie Mellon University

July 28, 2000

(2000) *Intuitionistic Reasoning about Shared Mutable Data Structures*

- Not the final version of separation logic, but introduces the separating conjunction.

This paper came like a bolt from the blue.—Brookes, O'Hearn, and Reddy

by Palgrave. It supercedes the version (dated August 12, 1999) that was distributed at the meeting, which contained a serious error.

Intuitionistic Reasoning
about Shared Mutable Data Structure*

John C. Reynolds
Department of Computer Science
Carnegie Mellon University

July 28, 2000

(2001) *Local Reasoning about Programs that Alter Data Structures*

- Canonical semantics and proof theory of separation logic.

Local Reasoning about Programs that Alter Data Structures

Peter O'Hearn[1], John Reynolds[2], and Hongseok Yang[3]

[1] Queen Mary, University of London
[2] Carnegie Mellon University
[3] University of Birmingham and University of Illinois at Urbana-Champaign

Abstract. We describe an extension of Hoare's logic for reasoning about programs that alter data structures. We consider a low-level storage model based on a heap with associated lookup, update, allocation and deallocation operations, and unrestricted address arithmetic. The assertion language is based on a possible worlds model of the logic of bunched implications, and includes spatial conjunction and implication connectives alongside those of classical logic. Heap operations are axiomatized using what we call the "small axioms", each of which mentions only those cells accessed by a particular command. Through these and a number of examples we show that the formalism supports local reasoning: A specification and proof can concentrate on only those cells in memory that a program accesses.
This paper builds on earlier work by Burstall, Reynolds, Ishtiaq and O'Hearn on reasoning about data structures.

1 Introduction

(2001) *Local Reasoning about Programs that Alter Data Structures*

- Canonical semantics and proof theory of separation logic.

Local Reasoning about Programs that Alter

If it was a dirty program before, it's not anymore.—John Reynolds

[1] Queen Mary, University of London
[2] Carnegie Mellon University
[3] University of Birmingham and University of Illinois at Urbana-Champaign

Abstract. We describe an extension of Hoare's logic for reasoning about programs that alter data structures. We consider a low-level storage model based on a heap with associated lookup, update, allocation and deallocation operations, and unrestricted address arithmetic. The assertion language is based on a possible worlds model of the logic of bunched implications, and includes spatial conjunction and implication connectives alongside those of classical logic. Heap operations are axiomatized using what we call the "small axioms", each of which mentions only those cells accessed by a particular command. Through these and a number of examples we show that the formalism supports local reasoning: A specification and proof can concentrate on only those cells in memory that a program accesses.
This paper builds on earlier work by Burstall, Reynolds, Ishtiaq and O'Hearn on reasoning about data structures.

1 Introduction

(2002) *Separation Logic: A Logic for Shared Mutable Data Structures*

- Invited talk at LICS 2002.

Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of

depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

$$\mathsf{j} := \mathbf{nil}\ ;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$$
$$(\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).$$

(Here the notation $[e]$ denotes the contents of the storage at address e.)

The invariant of this program must state that i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of α onto β:

$$\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta,$$

(2002) *Separation Logic: A Logic for Shared Mutable Data Structures*

- Invited talk at LICS 2002.

Separation Logic: A Logic for Shared Mutable Data Structures

A growing number of people are working on this formalism and... well... we think we're on to something.—John Reynolds

Abstract

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of

depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

$$\mathsf{j} := \mathbf{nil}\ ;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$$
$$(\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).$$

(Here the notation $[e]$ denotes the contents of the storage at address e.)

The invariant of this program must state that i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of α onto β:

$$\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta,$$

Types, Abstraction and Parametric Polymorphism (Citations: 417) Export

Edit

John C. Reynolds

Conference: World Computer Congress - IFIP, pp. 513-523, 1983

Separation Logic: A Logic for Shared Mutable Data Structures (Citations: 779) Export

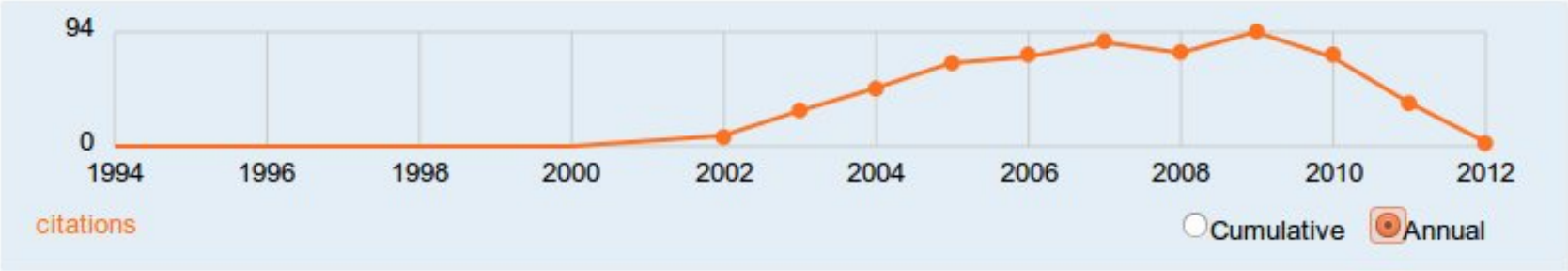
Edit

John C. Reynolds

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of **Hoare logic** that permits reasoning about low-level impera- tive programs that use shared mutable data structure. The simple imperative **programming language** is ex- tended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation a nd deallocation of storage. Assertions are extended by intro- ducing a "separating conjunction" that asserts that its sub - formulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the induc- tive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing. In this paper, we will survey the current development of this program logic, including extensions that permit unre- stricted address arithmetic, dynamically allocated array s, and recursive procedures. We will also discuss promising future directions.

Conference: Logic in Computer Science - LICS, pp. 55-74, 2002

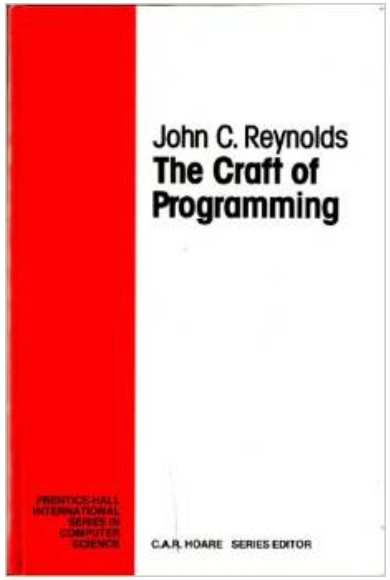
DOI: 10.1109/LICS.2002.1029817

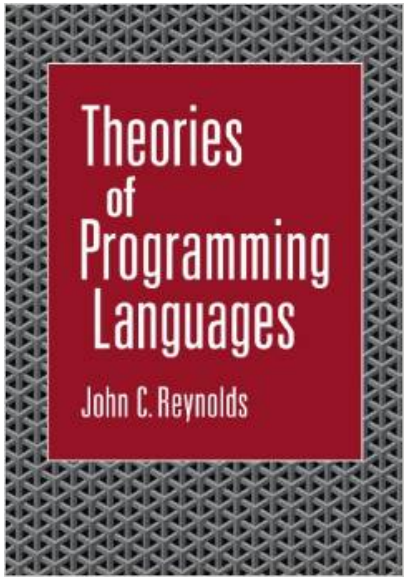


Books

(1981) *The Craft of Programming* (Prentice Hall)

(1998) *Theories of Programming Languages* (Cambridge University Press)

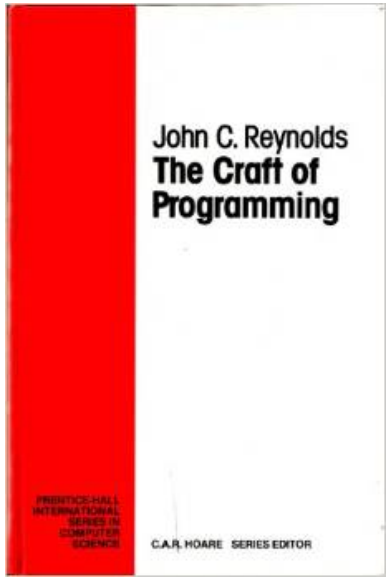


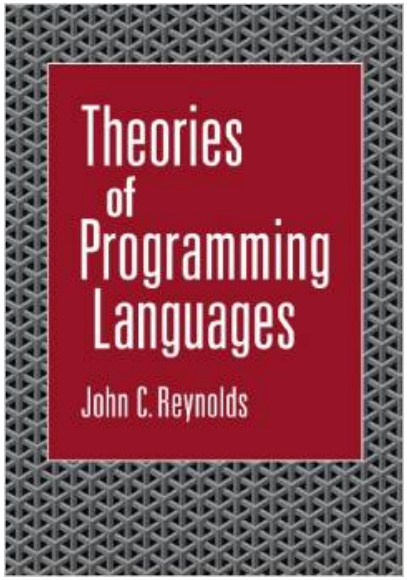


Books

(1981) *The Craft of Programming* (Prentice Hall)

(1998) *Theories of Programming Languages* (Cambridge University Press)





`http://repository.cmu.edu/compsci/1280/`

Awards

(2010) Lovelace Medal (British Computer Society)

(2007) Honorary D.Sc. Degree (Queen Mary, University of London)

(2006) Dana Scott Distinguished Research Career Award (Carnegie Mellon University)

(2003) SIGPLAN Programming Language Achievement Award

(1971) ACM Annual Programming Systems and Languages Paper Award (for paper on GEDANKEN)

(1953) National Science Talent Search

Awards

(2010) Lovelace Medal (British Computer Society)

(2007) Honorary D.Sc. Degree (Queen Mary, University of London)

(2006) Dana Scott Distinguished Research Career Award (Carnegie Mellon University)

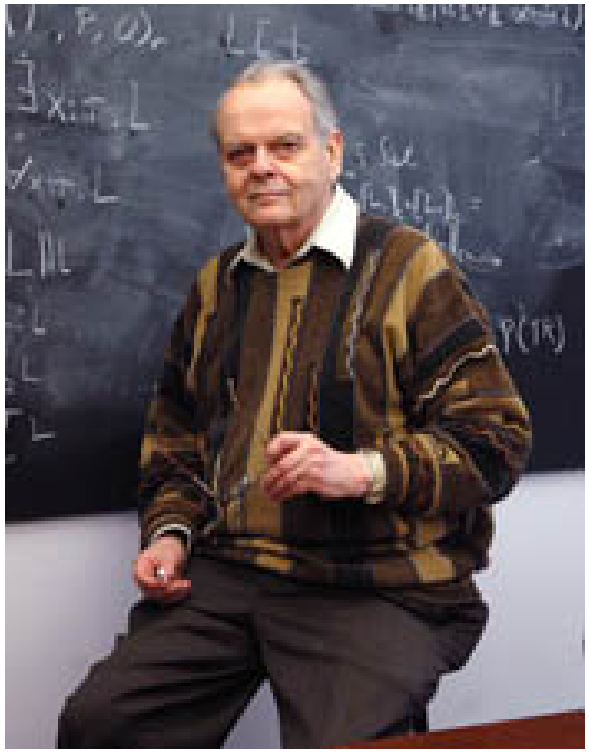
(2003) SIGPLAN Programming Language Achievement Award

(1971) ACM Annual Programming Systems and Languages Paper Award (for paper on GEDANKEN)

(1953) National Science Talent Search

In his honour, the ACM SIGPLAN renamed the Oustanding Doctoral Dissertation Award to the John C. Reynolds Doctoral Dissertation Award.

Why was Reynolds so great? Well, he was brilliant and a hard working. . . but one thing that was special was how he interacted with colleagues. I met him in 1989, as a PhD student. Working on specification logic, I was sent to CMU to talk to John. John didn't know who I was. I stayed at his house, and I started explaining my ideas. It wasn't going very well: I was explaining things in an over-complicated fashion. John never got angry, nor gave me a hard time, but kept asking questions. After four hours of questioning, he slapped his forehead and said, "now I see what you're up to". He didn't care about fashion, funding, fame, credit; all he cared about was ideas. When he learned a new idea, he was very happy. And you could see it.—Peter O'Hearn



References I

Brookes, S., O'Hearn, P. W., and Reddy, U. S. (2014).
The essence of Reynolds.
Formal Aspects of Computing, 26(3):435–439.

C.Reynolds, J. (1972).
Definitional interpreters for higher-order programming languages.
In *Proceedings of the 25th National ACM Conference*, volume 2, pages 717–740.

O'Hearn, P., Reynolds, J. C., and Yang, H. (2001).
Local reasoning about programs that alter data structures.
In *Proceedings of the International Workshop in Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19.

Reynolds, J. C. (1965).
An introduction to the COGENT programming system.
In *Proceedings of the 20th National ACM Conference*, pages 422–436.

References II

Reynolds, J. C. (1968).
Automatic computation of data set definitions.
In *Proceedings of Information Processing*, volume 1, pages 456–461.

Reynolds, J. C. (1970a).
GEDANKEN - a simple typless language based on the principle of completeness and reference concept.
Communications of the ACM, 13(5).

Reynolds, J. C. (1970b).
Transformational systems and the algebraic structure of atomic formulas.
In *Machine Intelligence*, pages 135–151.

Reynolds, J. C. (1974).
Towards a theory of type structure.
In *Paris colloquium on programming*, volume 19 of *LNCS*.
Springer-Verlag.

References III

Reynolds, J. C. (1975).
User-defined types and procedural data structures as complementary approaches to data abstraction.
In *Conference on New Directions in Algorithmic Languages*, pages 157–168.

Reynolds, J. C. (1981a).
The craft of programming.
Prentice-Hall.

Reynolds, J. C. (1981b).
The essence of Algol.
In *International Symposium on Algorithmic Languages*, pages 345–372.
Reprinted in 1997 in *Algol-like languages*, 1, 67-88.

Reynolds, J. C. (1983).
Types, abstractions, and parametric polymorphism.
In *Proceedings of Information Processing*.

References IV

Reynolds, J. C. (1995).
Using functor categories to generate intermediate code.
In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, pages 25–36.

Reynolds, J. C. (1998).
Theories of Programming Languages.
Cambridge University Press.

Reynolds, J. C. (2000).
Intuitionistic reasoning about shared mutable data structure.
In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in honour of Sir Tony Hoare*.

Reynolds, J. C. (2002).
Separation logic: A logic for shared mutable data structures.
In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74.