The Essence of Reynolds

Stephen Brookes
Carnegie Mellon University

Peter W. O'Hearn
Facebook

Uday Reddy
University of Birmingham

Abstract

John Reynolds (1935-2013) was a pioneer of programming languages research. In this paper we pay tribute to the man, his ideas, and his influence.

Categories and Subject Descriptors D.3 [*Programming Languages*]; F.3 [*Logics and Meanings of Programs*]

Keywords Polymorphic λ-calculus, Separation Logic, Parametricity, Data Abstraction, Subtypes, Defunctionalization, Substructural Logic and Typing, Possible World Semantics

1. Introduction

John Reynolds died on April 28, 2013. He will be remembered for his fundamental and far-sighted research on programming languages: semantics, specifications, language design, logics and proof methodology.

John was born on June 1, 1935, and was raised in Glen Ellyn, Illinois, a suburb of Chicago. He attended Purdue University as an undergraduate, graduating in 1956. His Ph.D. thesis in Theoretical Physics (Harvard University, 1961) was titled "Surface Properties of Nuclear Matter". In later years, with typical modesty and humility, John would describe this, one of his earliest experiences in computer science, as a "big number-crunching program" designed to produce "an uninteresting computation of an unimportant quantity in a bad approximation". At Harvard John met Mary A. Allen, and they married in 1960. John is survived by Mary and their sons, Edward (born 1963) and Matthew (born 1965).

After Harvard, with a burgeoning interest in computer science, John worked as a physicist at Argonne National Laboratory. While at Argonne, he developed a compiler for compilers, called COGENT (1965), and he designed the language GEDANKEN (1969). In 1970 he left Argonne for academia, moving to Syracuse University as a Professor of Computer and Information Science. After a long and fruitful period at Syracuse he moved to Carnegie Mellon University in 1986, where he stayed until retirement at the end of 2012. Over the years he enjoyed sabbaticals and visiting positions at many universities and institutions, including Stanford University, INRIA (Rocquencourt and Sophia Antipolis), Imperial College (University of London), Lucent Technologies, Queen Mary (University of London), Edinburgh University, Aarhus University, and Microsoft Research (Cambridge, England). He forged long-lasting ties with many researchers and was highly valued as a colleague. He was an invited speaker at conferences and symposia spanning the world.

John was an active member of the Association for Computing Machinery (ACM) for many years, including a long stint as editor of the Communications of the ACM and the Journal of the ACM. He became an ACM Fellow in 2001, and received the ACM SIGPLAN Programming Language Achievement Award in 2003. He was a prominent member of IFIP Working Group 2.3. (Programming Methodology), which he joined in 1969, and IFIP Working Group 2.2 (Formal Language Definition), in which he participated from 1977 to 1991. The British Computer Society awarded him the Lovelace Medal in 2011. He was a proud recipient of an Honorary D. Sc. degree conferred by Queen Mary, University of London, in July 2007. Carnegie Mellon University honored him with the Dana Scott Distinguished Research Award in April 2006. In November 2013 ACM SIGPLAN honoured John's memory by renaming their Outstanding Doctoral Dissertation Award to the John C. Reynolds Doctoral Dissertation Award.

John laid the foundations for the work of many of us in the programming languages research community. The influence of his more widely known work – on parametric polymorphism and on Separation Logic – is very plain to see, for example by browsing the pages of recent POPL conferences. What is less well known, and perhaps even 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. In remembering him, and what he was like as a researcher and colleague, there is no better place to start than with a simple recollection of these results.

2. Main works.

Early Works. While working at Argonne John became interested in both programming languages and in automatic theorem proving. He produced COGENT, an early compiler-compiler. He then produced a very significant, yet somewhat overlooked, paper:

- **Automatic computation of data set definitions**. IFIP Congress (1) 1968: 456-461

In modern terminology, this paper computed an over-approximation of the kinds of data structures reached in a program, where the data was constructed using Lisp-style operations (car, cdr, etc).

There is a story associated with this paper. Years later, people were pursuing automatic shape analysis based on Separation Logic, as represented in tools such as SPACE INVADER, SLAYER, THOR and INFER. Reynolds wanted to pursue proofs of full functional correctness, beyond the reach of mere program analysis, leading to many good-natured arguments and discussions. In arguing his position, John seemed to be eerily skilled at pinpointing the limitations of the techniques in these tools, even though he could see and appreciate the excitement. It turns out that the people working on these tools were not aware that "Automatic Computation of Data Set Definitions" could be regarded as an early shape analysis.

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2535838.2537851

You can find yourself at a disadvantage when your opponent just happens to be the source of the founding works underpinning the position you are defending!

This story is typical. Again and again during his career Reynolds was seen to be years ahead of the field, and it became a kind of joke amongst friends that John had gotten there 10 years or more before everyone else. Once, when hearing a talk on types, Reynolds complimented the speaker on one of the ideas he explained, and the speaker said that the idea was taken from a Reynolds paper. At this point a round of jokes began on the theme of Reynolds being 10 years ahead of himself; John joined in, teasing himself gleefully.

The next important paper was in the direction of John's other early interest, theorem proving.

- **Transformational systems and the algebraic structure of atomic formulas**. Machine Intelligence 5, pages 135151. 1969.

This paper developed an anti-unification algorithm, which solves the dual problem to unification: it seeks the most specific common generalization of two terms. This is a fundamental idea that forms part of the basis of the field of inductive logic programming.

These two papers, which are probably not so well known in the PL community, would already have given a very respectable career. But it was now that John entered a kind of golden of period, an extended run of high productivity and extreme quality.

Golden Period. 1970-1984. There are so many top-flight papers here that we simply list them with brief comments, a kind of annotated bibliography.

- **GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept**. CACM 13(5): 308-319 (1970)

  This was an untyped call-by-value programming language whose way of mixing imperative features and functions is similar to the core of ML. This paper won the 1971 ACM Annual Programming Systems and Languages Paper Award, 1971

- **Definitional Interpreters for Higher-Order Programming Languages**, Proceedings, 25th National ACM Conference (August 1972), pp. 717-740. Reprinted in Higher-Order and Symbolic Computation, 11 (1998), pp. 363-397.

  This paper showed how to transform a series of interpreters, eventually ending up with a compiler. This work helped to popularize continuations as a tool in language implementation, and introduced the concept of *defunctionalization* where functions in the interpreter are replaced by their representations (such as closures) in more concrete data structures.

- **On the Relation between Direct and Continuation Semantics**. ICALP 1974 : 141-156

  At the time the power of continuations was becoming apparent, and John asked the simple question of whether continuation and direct semantics can be connected by a precise theorem. The question was non-trivial because, as the abstract states, 'The heart of the problem is the construction of a relation which must be a fixed-point of a non-monotonic 'relational functor".' The paper introduced a general technique for constructing relations between recursively-defined domains.

- **Towards a theory of type structure**. Symposium on Programming 1974: 408-423

  Introduced the Polymorphic λ-calculus, the first type theory of polymorphic functions. Reynolds's work on Polyλ as well as his later work on parametricity influenced the designs of generics in Java and in .Net (Phil Wadler and Andrew Kennedy, personal communication). The polymorphic lambda calculus is beautiful as well as powerful. It consists of just five constructs, yet it can define many of the data structures of central importance in programming. Like many timeless scientific discoveries, this calculus was invented twice, in this case by Reynolds and by the logician Jean-Yves Girard (who called it System F).

  In this paper John attempted, not for the last time, to make a model of the polymorphic calculus. This was difficult because of the impredicative (circular) nature of the calculus, which made it seem to be skirting Russell's paradox. He almost got there, but stated: "we must admit a serious lacuna in our chain of argument". Later, John said (jokingly) that the lacuna admission was one of the most fortunate phrases he had ever included in a paper.

- **User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction**, New Directions in Algorithmic Languages 1975

  This paper was ahead of its time in pinpointing the essential difference between object-oriented approaches to data abstraction and abstract data types... a topic often revisited decades later. Essentially, procedural approaches are easy to extend because their data representations are decentralized, ADT's provide for binary and multi-nary operations because they are centralized, and the strengths of one are the limitations of the other.

- **Syntactic Control of Interference**. POPL 1978: 39-46

  A strikingly original paper on using what is now called an affine λ-calculus to control aliasing and other forms of interference between program parts. Possibly the first use of substructural type theory to control resources, 9 years before the appearance of linear logic.

- **Programming with Transition Diagrams**. Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3, ed. D. Gries, Springer-Verlag, 1978, pp. 153–165.

  In this paper, Reynolds goes against the grain, demonstrating how well-designed programs with goto statements can be easy to prove correct.

- **Reasoning about Arrays**, CACM 22(5): 290–299 (1979)

  An early application of *separation* principles for reasoning about arrays using a notation called partition diagrams that Reynolds invented for this purpose. The material is also covered in Chapter 2 of *Craft of Programming*. A technique that Reynolds often used for motivating this material was to ask students to write a binary search program and let them discover for themselves how it is hard it is to get it right. With Reynolds's technique, it is almost impossible to get it wrong.

- **Using category theory to design implicit conversions and generic operators**, Semantics-Directed Compiler Generation 1980: 211-258

  A generalization of many-sorted algebras, category-sorted algebras, is defined and used to account for generic operators applied to different types in the presence of implicit conversions between types, ensuring that the order of applying the conversions does not affect the final result.

- **The Essence of Algol**, Algorithmic Languages, ed. J. W. de Bakker and J. C. van Vliet, North-Holland, 1981, pp. 345-372.

  In this highly influential and oft cited paper, Reynolds contended that Algol should be considered as an applied call-by-name λ-calculus, and described Idealized Algol to illustrate a number of principles of language design that he proposed as constituting Algol's essence. This paper made significant technical inventions that go far beyond Idealized Algol.

  - Explained how types encapsulate effects naturally in the call-by-name design, and consequently full $\beta\eta$ laws are valid. This idea was echoed in the use of types to control effects in Haskell, which used similar principles to insulate

functions from effects and went beyond Algol's monoid of commands to monads (a sort of parameterized commands).

- Possible world (functor category) semantics of local state, which is still influencing technical developments at the edge of programming language theory, as in recent POPL papers of Ahmed, Dreyer, Birkedal and others on reasoning about local state with Kripke relations.
- Typed λ-calculus with subtypes, which has been used significantly in the foundations of object-oriented languages and elsewhere.

Typically, John did not make a big deal about any of these specific contributions, or single out any one for special attention; it was as if they were mere intellectual tools used in crafting his larger points about language design. But what tools! One wonders in retrospect how he managed to cram so many significant and original ideas into a single paper.

- **The Craft of Programming**. Prentice Hall International series in computer science, Prentice Hall 1981, ISBN 978-0-13-188862-3, pp. I-XIII, 1-434
  and
  Idealized Algol and its Specification Logic, Tools and Notions for Program Construction, ed. D. Neel, Cambridge University Press (1982), pp. 121-161.

  In this book and paper Reynolds's ideas about program proving are developing. There is a significant focus on abstraction and refinement in "craft", and on controlling interference as an aid to modular reasoning.
- **Types, Abstraction and Parametric Polymorphism**. IFIP Congress 1983: 513-523

  Perhaps the only computer science paper in which the phrase 'moral turpitude' features, in a not-to-be-missed fable involving Bessel and Descartes. Arguably the best article ever written on the point and purpose of types, exploring the idea that types enforce abstraction. Written as a counterpoint to the prevailing attitude at the time, in which types (only) classify values and rule out errors; while the conventional view emphasized one laudable aspect, Reynolds was convinced that there was more to types and makes a compelling case.

  Technically, the paper is notable for formalizing the Abstraction Theorem and for introducing Relational Parametricity.
- **Polymorphism is not Set-Theoretic**. Semantics of Data Types 1984: 145-156

  Reynolds had been convinced that the polymorphic λ-calculus should have a set-theoretic model, because "types are not limited to computation", and he set out to find a set-theoretic model. Instead, he ended up showing that there actually is no non-trivial model of the polymorphic λ-calculus in which types denote sets and where the function type denotes the collection of all set-theoretic functions from one type to another. Interestingly, this was done assuming *classical* set theory, and observations of Hyland, Moggi, Pitts and others later demonstrated the possibility of set-theoretic models assuming *intuitionistic* set theory. Nowadays there are many models of the polymorphic calculus.

  Although he did not question the truth of his result, John was never satisfied with the models of polymorphism that have been obtained, and upon questioning he would become fidgety: it was as if he felt there was something missing in his own insight (if not that of others).

1985-2000. In this period John continued producing high-quality work, but not at quite the same pace as in the golden period. Some of the key works are the following.

- **Syntactic Control of Interference, Part 2**, ICALP'89
- **On Functors Expressible in the Polymorphic Typed Lambda Calculus**. Inf. Comput. 105(1): 1-29, 1993 (with Plotkin)
- **Using Functor Categories to Generate Intermediate Code**. POPL 1995: 25-36
- **Theories of programming languages**. Cambridge University Press 1998, ISBN 978-0-521-59414-1, pp. I-XII, 1-500
- **From Algol to polymorphic linear lambda-calculus**. J. ACM 47(1): 167-223, 2000. (with O'Hearn)

The JACM'00 paper connected up his interests on Algol and on polymorphism; more on that in the next section.

The POPL'95 paper is remarkable for connecting category theory to compilation. John used category theory, and semantics in general, as a tool for guiding design (of a language or a compiler), and not just for after-the-fact study. This attitude is summed up neatly in his remark:

> *Programming language semanticists should be the obstetricians of programming languages, not their coroners.*
> John C. Reynolds

Separation Logic. In 1999, at age 64, John produced one of his most stunning papers.

- **Intuitionistic Reasoning about Shared Mutable Data Structure**, Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare, Palgrave, 2000.

This paper came like a bolt from the blue. In it, John showed proofs of heap-mutating programs that were almost as simple as those for purely-functional programs. The key insight was how linked data structures could be described using inductive definitions and a connective which we now call the separating conjunction. One could fold and unfold the definitions, and work in isolation on components of the separating conjunction.

John's paper built on an early work of Burstall (Some techniques for proving programs which alter data structures. Machine Intelligence, 1972). Reynolds extended it by using a form of substructural logic, which turned out to be an instance of the Bunched Logic of O'Hearn and Pym. After a POPL'01 paper by O'Hearn and Ishtiaq detailing this connection, there was one further paper.

- **Local Reasoning about Programs that Alter Data Structures**. CSL 2001: 1-19 (with O'Hearn and Yang)

This paper described the canonical semantics and proof theory of Separation Logic as it is now understood. It used a boolean rather than an intuitionistic form of logic and it worked for a low-level programming language closer to assembly, whereas the 1999 paper was formulated for a language without memory deallocation or address arithmetic.

Although Reynolds was a proponent of influential ideas, he never regarded the positions he adopted as cast in stone. Indeed, it might seem strange that John Reynolds, one of the leading proponents of safe programming languages, would advance a logic for an unsafe language in his work. Here was the inventor of GEDANKEN, Idealized Algol, and the Polymorphic λ-calculus, working with a language with 'dirty' features. But, armed with program logic, safety could be achieved. Reynolds proved a number of low-level programs to test his ideas, such as for a form of doubly-linked lists which save space by storing the xor of forward and back pointers rather than each separately, and the inductive definition in Separation Logic and the proof came out surprisingly easily.

Next, John wrote an influential survey article covering these papers and others on Separation Logic up to 2002.

- **Separation Logic: A Logic for Shared Mutable Data Structures**. LICS 2002: 55-74

This paper was associated with an invited talk by John at the LICS'02 conference. At the end of his talk John started to try to sum up where we had got to and then stopped, looked at the audience, and simply said: 'a growing number of people are working on this formalism and... well... we think we're on to something'. John Reynolds was humble and not prone to overstatement, and coming from him this was a strong statement.

The next section will provide more context for this work on Separation Logic.

Later Papers. After the founding of Separation Logic, John produced several further papers.

- **Separation and information hiding**. POPL 2004: 268-280 (with O'Hearn and Yang)
- **Local reasoning about a copying garbage collector**. POPL 2004: 220-231 (with Birkedal and Torp-Smith)
- **Towards a Grainless Semantics for Shared Variable Concurrency**, FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, LNCS 3328, pp. 35-48, 2004
- **Syntactic Control of interference for Separation Logic**. POPL 2012: 323-336 (with Reddy)

The paper on grainless semantics picks up an old bugbear of his, influenced by Dijkstra's idea that the understanding of concurrent programs should not be dependent on a particular 'grain to time'. His final paper, with Reddy, fittingly brings together two of his major contributions.

3. State and abstraction

The issues of state and abstraction were central themes in Reynolds's research, especially during the time of his "golden period." *State* is of the kind represented in imperative "Algol-like" languages, with Algol W being his favorite. This preference was because the effects are concentrated in the base types of "expressions" and "commands," and the function types remain "pure," subject to full $\beta\eta$ equivalence. Thus, "Algol-like" meant an integration of "imperative" and "functional" aspects, preserving both of their fundamental properties without any corruption. *Abstraction* is represented in strong type systems, semantically based, as well as in how the store is viewed in Algol-like languages. Reynolds saw intimate connections between these two views of abstraction and sought to build bridges so that the ideas could be carried back and forth between them. In addition to the abstraction inherent in the store, Reynolds also wanted additional *modularity* in the management of store, going beyond what is representable in conventional language designs. This desire led him to new proposals for type systems, logics and semantic frameworks, as represented in *syntactic control of interference*, *Separation Logic* and *grainless semantics*.

Reynolds's golden period coincides with his teaching of a graduate course on programming at Syracuse. The material he developed for this course is published as *The Craft of Programming* (1981). It contains a wealth of information about

- what imperative programming *means*,
- how to *develop* imperative programs rigorously,
- the *type structure* of imperative programs,
- *reasoning principles* (both *practical*, *e.g.*, arrays, and *abstract*, *e.g.*, Specification Logic), and
- how to reason about *data abstraction* (or *information hiding*).

His analysis of this material gave rise to a series of insightful papers, the most prominent of which are *The Essence of Algol* (1981), and *Idealized Algol and its Specification Logic* (1982). However, the most novel material of the book, the chapter on *Data representation structuring*, is not directly represented in these papers. So, one must go to the source to understand these ideas. However, we believe that these insights influenced Reynolds's papers on types: *A theory of type structure* (1974) and *Types, abstraction and parametric polymorphism* (1983). These are the "bridges" we speak of.

Reynolds always thought of the store in imperative programs as an "abstract type," whose full information is hidden from the program. Soon after defining the polymorphic lambda calculus in 1974, he worked on a project titled *A polymorphic model of Algol*. We have a copy of his handwritten notes, dated "9–23–75," which contain these key equations

$$\begin{array}{rcl} B_{\mathbf{sta}}[S] & = & S \stackrel{\sim}{\to} S \\ B_{w_1 \to w_2}[S] & = & \forall S'.\, B_{w_1}[S \times S'] \to B_{w_2}[S \times S'] \end{array}$$

We see here that Reynolds thinks of the (Idealized) Algol types for statements (**sta**) and procedures ($w_1 \to w_2$) as *type constructors* $B_{\mathbf{sta}}$ and $B_{w_1 \to w_2}$. These type constructors are parametrized by a type variable S representing the set of states for a store. This is in marked contrast to Strachey's denotational semantics where the store is fixed and *global*. Here the store is a *type variable*. In essence, this means that the program can work with whatever "type" of store we supply, as long as it has all the required data variables. An intuitive interpretation of parametric polymorphism immediately implies that the program will act *the same way* for all these different "types" of stores.

This model of Algol could not be finished in 1975 because it did not satisfy the β equivalence of Algol. In 1981, Reynolds moved from type constructors to functors in the *Essence of Algol* model, which depend on the postulation of morphisms between store types. We suspect that he was never fully satisfied with functors. In 2000, he returned to the original equations in the joint paper with O'Hearn, *From Algol to polymorphic linear lambda calculus*, where these equations appear again, adapted to the linear context. So, what happened to the problem with β equivalence? It is still open! Linearity rules out the obvious counterexamples and enough evidence is gathered to show that the model is worthy even if it does not satisfy β equivalence.

This is one direction of the "bridge." We expect that the other direction was implicit in Reynolds's thinking. The chapter on *Data representation structuring* contains programming examples where graph algorithms are treated (including Tarjan's algorithm for strong components, posed as a challenge problem by Knuth for program correctness) using data refinement. An initial algorithm expressed in terms of sets of nodes and edges is gradually refined to an algorithm involving concrete data objects within Algol W. The relationship between the two sets of data structures is captured using a technique that Reynolds invents for this purpose: *general invariants*, which are true throughout a particular region of the program (as opposed to Floyd-Hoare invariants that are true at particular points in the program). Transporting these ideas back across the bridge, we notice that the programs are operating on two independent state spaces S_A and S_C, and the general invariants are relations $R \subseteq S_A \times S_C$ that are maintained by the two slices of the program without any observable difference. Here, we see the principles of relational parametricity taking shape, not in an abstract theoretical setting, but in concrete programming examples involving data abstraction.

The essence of relational parametricity is that, if a program term is parametrized by a type variable (or an *opaque* type), then the information about the type is hidden from the term, and the term behaves *the same way* for all possible instantiations of the type vari-

able. Hence, if we consider two instantiations A and A' of the type variable and allow ourselves the freedom to substitute particular elements $x \in A$ by elements $x' \in A'$ then the results produced by the program term should be the same, modulo the substitution we have made. What kind of substitutions can one make? In early years of modern mathematics, isomorphisms, *i.e.*, structure-preserving *one-to-one* correspondences, were allowed. After the advent of category theory, homomorphisms, *i.e.*, *many-to-one* correspondences, were allowed. But, if A and A' are two *arbitrary* instantiations, there is no intuitive reason why the allowed correspondences should be asymmetric. Hence, logical relations, *i.e.*, structure-preserving *many-to-many* correspondences, are the only intuitively appealing answer to the question for programming applications. They form the basis of *relational parametricity*. In a good theory of parametricity, both "isomorphism parametricity" and "homomorphism parametricity" (natural transformations) become special cases of relational parametricity.

While we mentioned "program terms" above for the sake of explanation, these ideas are by no means limited to programs. Any mathematical formula, any algorithmic process or any mechanism in which output quantities are causally derived from input quantities, whether discrete or continuous, can have the notion of being parametrized by types and, if so, we might expect it to be parametric. Bob Atkey makes a beginning with this line of enquiry in the present POPL. He shows that the symmetries of physical system as required for the "Noether's theorem" in classical mechanics can be viewed as instances of parametricity. This should pave the way for broader applications of the relational parametricity principle in mathematics and science.

With the formulation of relational parametricity, the "bridge" is completed. The reasoning principles of *Data representation structuring* are translated into the parametricity principles of the polymorphic lambda calculus, via the translation of the 1975 notes (or the 2000 paper employing polymorphic linear lambda calculus).

Imperative programs are complicated for a variety of reasons and those complications do not go away just by defining semantics. Reynolds repeatedly sought to create intellectual tools to control and master the complications. A key idea was that the state should not be treated as a monolithic entity; rather it should be partitioned into *separate* portions and distributed to different parts of the program to operate on independently. In *Syntactic Control of Interference* (1978), he proposed a system of rules that allows program terms to be viewed as "owning" portions of the store, based on their free identifiers, and only terms that own separate portions can be combined using the procedure mechanism. This is roughly the meaning of saying that a procedure and its argument "should not interfere." Specification logic (1982) involves subtler notions of non-interference, which were investigated in a novel semantics of Tennent (*Semantical Analysis of Specification Logic*, Inform. Comput. 1985) and in O'Hearn's 1991 Ph.D. thesis. In *Reasoning about Arrays* (1979), Reynolds applied the separation principle to arrays, inventing a powerful notation called "partition diagrams" to document such separation.

All of this was perhaps only setting the stage for the powerful *Separation Logic* for dynamic data structures to come in 1999-2001 which came as a surprising *tour de force* even to those of us that had thought about the problem and knew all the technical tools that Reynolds, O'Hearn and others deployed in solving it. Separation Logic represents a giant intellectual leap in recognizing that ownership (what can be accessed) and separation (divisions of the state) are not static, spanning all states, but can vary from state to state.

In his 2000 paper *Intuitionistic Reasoning about Shared Mutable Data Structure*, Reynolds introduced the separating conjunction, which allows the *pre-conditions* of operations to break the state into components. Reynolds demonstrated that this allowed for simple proofs of heap-mutating programs and, more deeply enabled a new spatial dimension to modular reasoning which complements that of abstraction. In a sense, the pre-conditions not only specify what is *true* in a state but also describe the *structure of the store* (mainly the heap). This idea has far-reaching consequences, and it is extremely fortunate that Reynolds made this step forward at the time he did, because he opened the flood gates for new ideas.

The foremost of the new ideas was the resolution of the frame problem by O'Hearn. In *Local reasoning about programs that alter data structures* (2001, by Reynolds jointly with O'Hearn and Yang), pre-conditions specify not merely what portion of the store *can be* used from a program point, but *only* that portion of the store that will be so used. This has the remarkable consequence that specifications need to talk about only the portion of the store actually manipulated by program fragments, referred to as their "footprint". A general rule called the *frame rule* allows such local specifications to be extended to larger contexts as needed; and, using the separating conjunction, this passage from smaller to larger contexts can be described extremely directly and compactly.

Separation Logic, as represented in the 'Local Reasoning' paper and finally in Reynolds's LICS'02 paper, can be seen as the culmination of his theme of *making the structure of the store explicit* in programming language theories (be they semantics or logics). In the end, Separation Logic derives much of its power from the interplay between dynamic separation and framing. And, the passage from local specifications to larger contexts by framing can be seen as a logical analogue of Reynolds's idea of expanding a store shape in his work on possible world semantics, going back as far as the 1975 notes mentioned above, but with the separating conjunction providing a more flexible form of extension.

Today, Separation Logic is a thriving research direction with a tremendous investment in building automated tools and solving a variety of reasoning problems that have been perceived as too difficult or impossible for a long time. It represents a major force for taming the looming problem of concurrent software to be run on multi-core processors.

4. Conclusion

John Reynolds was one of the greatest minds programming research has seen. What was his secret? What was the essence of Reynolds?

Certainly, he had great talent, and he also had great determination. But his way of working and interacting with others, conveying what was important to him, was special and inspiring to those who knew him. John Reynolds was as pure a scientist as you would ever meet, and this is what made him tick. It was not academic politics, or funding, or fame, it was just the ideas.

It was not uncommon for John to talk to someone about their own ideas for hours at a time. Not only 20 minutes, or one hour, but three, four or five hours. This was especially true for students and junior faculty, but also held for more senior people as well. He would set everything else aside, his own time included, in his desire to get to the heart of the matter. These were intense sessions of deep concentration and back-and-forth discussion, and while John would argue vigorously and ask probing questions he always managed to be patient and pleasant with whoever he was talking to. It was as if programming and programming languages held wonderful treasures waiting to be uncovered, and he wanted to see them. John Reynolds just wanted to understand.

And he came to understand a lot. In the course of his career he discovered numerous deep ideas of lasting value – anti-unification, defunctionalization, theory of subtypes, possible world semantics of state, polymorphic λ-calculus, relational parametricity, Separation Logic, and others – and we are much the richer for it.