Life and Work of Dr. Robin Milner

By:

Dr. Navin Kabra, Co-Founder and CTO, ReliScore.com
Robin Milner

Turing Award 1991 For

- LCF – Automated Theorem Proving
- ML – Programming Language
  - With Polymorphic Type Inference
  - Type-Safe Exception Handling
- CCS – A General Theory of Concurrency
A key ingredient in all of his work has been his ability to combine deep insight into mathematical foundations of the subject with an equally deep understanding of practical and aesthetic issues, thus allowing the feedback of theory into practice in an exciting way.
Outline

- Importance of Types
- Polymorphic Types
- Type Inferencing
- ML
- Life and Work of Robin Milner
Disclaimers

• Not an Expert
• Brief Overviews Only
• High-level Handwaving
**PYTHON:**

```python
def total(collection):
    tot = 0
    for item in collection:
        tot += item
    return tot
```

**JAVA:**

```java
import java.util.*;

class Main {
    public static int total(List collection) {
        int tot = 0;
        Iterator it = collection.iterator();
        while (it.hasNext()) {
            Integer item = (Integer)it.next();
            tot += item;
        }
        return tot;
    }
}
```
Java is a typed language
Python has no types (???)
Java is a statically typed language
Python is dynamically typed
Java: types checked at compile-time

Python: types checked at runtime
**PYTHON:**

def total(collection):
    tot = 0
    for item in collection:
        tot += item
    return tot

**JAVA:**

import java.util.*;

class Main {
    public static int total(List collection) {
        int tot = 0;
        Iterator it = collection.iterator();
        while (it.hasNext()) {
            Integer item = (Integer)it.next();
            tot += item;
        }
        return tot;
    }
}
**PYTHON:**

def total(collection):
    tot = 0
    for item in collection:
        tot += item
    return tot

**JAVA:**

import java.util.*;

class Main {
    public static int total(List<Integer> collection) {
        int tot = 0;
        Iterator<Integer> it = collection.iterator();
        while (it.hasNext()) {
            tot += it.next();
        }
        return tot;
    }
}

Java: safer
Python: easier
The problem of generic computation
**PYTHON:**

def total(collection):
    tot = 0
    for item in collection:
        tot += item
    return tot

**JAVA:**

import java.util.*;

class Main {
    public static int total(List<Integer> collection) {
        int tot = 0;
        Iterator<Integer> it = collection.iterator();
        while (it.hasNext()) {
            tot += it.next();
        }
        return tot;
    }
}

Works for List of any type that supports '+'

Works for List of Integer only
import java.util.*;

class Main<T> {
    public static T total(List<T> collection) {
        T tot = 0;
        Iterator<T> it = collection.iterator();
        while (it.hasNext()) {
            tot += it.next();
        }
        return tot;
    }
}

**GENERIC Types in JAVA:**
**GENERIC Type Examples:**

```java
class MyList<T> {
    T[] array_of_items;

    public MyList<T>() {
        this.array_of_items = new T[10];
    }

    public T getElem(int n) {
        return this.array_of_items[n];
    }

    public void bubbleSort() {
        for (int i = 0; i < array_of_items.length; i++) {
            for (int j = 0; j < i; j++) {
                if (compare(array_of_items[i], array_of_items[j]) > 0) {
                    T tmp = array_of_items[i];
                    array_of_items[i] = array_of_items[j];
                    array_of_items[j] = tmp;
                }
            }
        }
    }
}
```
Type Inferencing
Java (static typing): safer

Python (dynamic typing): easier
Inferring Types

- All types need not be user-specified
- Compiler can infer types
int f(String s) {
    return s.string_length;
}

int g(String x, int y) {
    return f(x) + y;
}
f(String s) {
    return s.string_length;
}

g(x, y) {
    return f(x) + y;
}

------

Type inferencing

s.string_length is int hence return-type of f is int
f(x) is called, so x is String
f(x) returns int, so f(x) + y is int
Hence return-type of g is int
f(s) {
    return s.string_length;
}

g(x, y) {
    return f(x) + y;
}

------
Type inferencing

s is any type that allows s.string_length
Only String has s.string_length, so s is String
s.string_length is int
So f is function that takes String and returns int
f(x) returns int, so f(x) + y is int
Hence return-type of g is int
f(s) {
    return s.length;
}

g(x, y) {
    return f(x) + y;
}

Type inferencing

s is any type that allows s.length
Let's call that type S
s.string_length depends upon S
Let type of s.string_length be L
So f is function that takes S and returns L
f(x) + y, so L must support '+'
If only integers support '+' then L is int
Then f is function that takes S and returns int
g is a function that takes S, int and returns int
f(s) {
    return s.length;
}

g(x, y) {
    return f(x) + y;
}

-------

Type inferencing ... continued

More inferencing possible:
    if String is the only class that has s.length
    returning int, then S is String
Type Inferencing – Conceptual View

- Program consists of literals, variables and expressions
- Literals have fixed types
- Variables' types have to be inferred
Type Inferencing – Basic Idea

- Every expression imposes constraints on possible types of variables
- Find an assignment of types that satisfies all constraints
- “Unification” problem
  - Use union-find algorithm
Most General Type

- Case 1: x is int, y is List<int>, f is function taking int, List<int> and returning int
- Case 2: x is T, y is List<T>, f is function taking T, List<T> and returning T
ML
ML

- Milner's ML first programming language with polymorphic type inference
- Direct Influence on F#, Scala, O'Caml, Java
- Indirect influence on most modern programming languages
ML Features

- Impure functional programming language
  - Allows side-effects

- Features
  - Call-by-value (greedy)
  - First class functions
  - Garbage collection
  - Static typing
  - Parametric polymorphism
  - Polymorphic Type Inferencing
  - Algebraic Data Types
  - Pattern Matching
  - Exception Handling
ML
Solid Mathematical Foundation, but Practical Language
Why ML?
Life & Work of Robin Milner
Robin Milner – Early Life

- Born in UK – military family
  - Second Lieutenant in Royal Engineers
- 1950s: King's College Cambridge
  - B.Sc. Math & Philosophy
  - First exposure to programming
    - (Assembly!)
Programming was not a very beautiful thing. I resolved I would never go near a computer in my life.

Millner found the activity of "writing one instruction after the other" to be "inelegant" and "arbitrary".
Early Life - Contd

- No PhD
- 1 year as a school teacher
- 2 years of National Service – Suez
- Job for Computer Manufacturer Ferranti
  - QA, Program Library Maintenance, Compiler Upgrades
Entry to Academia

- 1963 City University London:
  - “moved to academia with a passionate belief that a deeper understanding of programming could form the basis for a more elegant practice.”

- Exposure to relational algebra
  - “but did not get very far with that”
Program Semantics

- 1968: Exposure to Christopher Strachey & Dana Scott
  - “Computable Functionals”
  - Abstract mathematical model for programs
    - Meaning of programs
    - “Denotational Semantics”

- Exposure to Provably Correct Programs
Automated Theorem Proving

- Program Verification
  - Manually intensive, boring, error-prone job

- Milner's idea: Use programs for automated proofs
  - Result: Toy programs only
  - Difficult to do anything interesting
Assisted Theorem Proving

- 1971-1973 Stanford
  - Exposure to John McCarthy's AI program
  - Milner's idea: assist human intelligence, don't replace it
  - Use machines to help humans do Strachey-Scott style proving
    - On practical examples
Stanford LCF

- Interactive user-driven proof system
- Computer given goals, sub-goals
- Computer constructs whole proof under user direction
- Reduces tedium, ensures correctness
- User does the high-level "thinking"
What's a Proof?

- Start with axioms
  - Known facts
- And inference rules
  - Method of generating new facts from old ones
- Repeatedly apply rules to prove theorems
Proof Assistant

- Understands axioms, inference rules
- Understands standard patterns of proof
  - e.g. substitution, induction, simplification
  - “Tactics”
- Understands how to combine multiple tactics
Applications of Theorem Proving

- Intel/Motorola use for chip verification
  - FDIV bug
Generalizing Theorem Proving

- How to add new tactics and new tacticals to a proof assistant
- How to prove that they don't have bugs
Meta Language

• Language to define:
  ♦ Axioms
  ♦ Inference Rules
  ♦ Patterns of Proof (Tactics)
  ♦ Composition of Tactics (Tacticals)

• How
  ♦ Data types, whose pre-defined values are axioms
  ♦ Operations on data-types are inference rules
ML Requirements - Broad

- Solve practical problems
- Use absolute minimum concepts
- Rigorously defined and analyzed
ML Requirements - Detailed

- Algebra based (not hardware)
- Very simple foundation
- Orthogonal ideas
  - Study and analyze independently
  - Combine in powerful ways
- Easy to define and prove semantics
ML Design

- Needs of theorem proving influenced design
- Narrow focus = simple/elegant language
- But (surprisingly?) wide applicability

Examples:
- Functional — functions = subgoals
- Exceptions — because proof strategies might fail
Problems with LCF

- Works only for simple sequential programs
- What about concurrent programs?
- What about user inputs
- What about non-deterministic behavior?
CCS

- 1980s
- Theoretical foundation for provable correctness of concurrent programs
- Modeling of shared data structures (memory locations)
After Turing Award

- Pi-calculus
- Theory of communicating and mobile programs
- Allow network topology to change during program execution
Impact of pi-calculus

- spi-calculus used in proofs of cryptographic protocols
- Stochastic pi-calculus used in molecular biology
- Also used in business processes
Challenges

- Need mathematical foundations for:
  - Data provenance
  - Privacy
  - Authorization
  - Trust (amongst users)
  - Self-awareness
  - Self-management
Milner died on 20\textsuperscript{th} March 2010
but his work lives on
A key ingredient in all of his work has been his ability to combine deep insight into **mathematical foundations** of the subject with an equally deep understanding of **practical and aesthetic issues**, thus allowing the feedback of theory into practice in an exciting way.