Programming Languages and Systems Literature Review

Emily Herbert

September 23, 2019

Contents

Chapter 1

Introduction

Accumulation of summaries and notes from various PL papers. This is endeavor is mostly for my own personal use, but I figured eh, open science is a good thing. So, a lot of what's written here may just be half-formed ideas, or notes that really only make sense to me.

Entry format is (sometimes) as follows.

Title of Paper [citation].

Description of contents.

- Side note.
- Clarifying question?

Chapter 2

Functional Programming

2.1 Data Structures

An Introduction to the Theory of Lists [2].

Formal definitions for lists and list operations. Discusses common list operations (map, filter, reduce, etc.) and offers best use cases. Provides examples of multiple interacting operations and operation equivalences.

- What are infinite lists?

Functional Pearl: The Zipper [7].

Describes a data structure that is akin to a zipper, for use in situations in which trees need to be modified non-destructively. Handles can be on particular elements of the tree, where locations hold the downward current subtree and the upward path. Functions are given for navigation left, right, up, and down, retrieving the nth element, changing an element in place, inserting elements left, right, and up, and deleting elements. A memoization approach is suggested, where "scars" hold tree structure for frequently visited elements.

- Scala implementation at https://github.com/stanch/zipper.
- Binary trees are shown, but binary tree example doesn't allow for any data stored in the tree?

Chapter 3

Object Orientated Programming

3.1 Java

The Expression Problem [13].
Proposes a solution to the expression problem - "goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code and while retaining static type safety." Solution in GJ creates a series of types that are dependent on one another, and relies on Interfaces for expression evaluation. Type variables can be indexed by any inner class defined in the variable's bound.

- I'm not sure I know what "indexing a type variable" is ?

Chapter 4

Machine Learning

4.1 Type Safety

Typesafe Abstractions for Tensor Operations (Short Paper) [4].

Typesafe tensors for tensor operations and machine learning. Tensors created with two types, the type (primitives) of the elements, and the phantom types used to label dimensions. This allows operations to be checked at compile time to ensure that all tensor operations are valid. Typesafe tensors are differentiable and can be type-checked. Computation graphs are also typed, and inputs/ outputs are also type-checked. Examples given for tensor operations - matrix multiplication, tensor contraction. Examples given for NN layers - FC, conv, recurrent (recursive).

Chapter 5

Simulation

5.1 Scala

Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation [12].
Proposes multi-agent simulation (MAS) in Scala that utilizes the Actor framework. Satisfies motivations for removing possibility of race conditions. Provides benchmarks comparing Actor framework to threads - Actor framework displays slower results, but the prospect of safer simulations justifies the slow-down.

- This design is outdated. What would a modern design look like?
- Would there be a motivation for simulation for RL agents?
- Scala's delimited continuations library.

Using Domain Specific Languages for Modeling and Simulation [10].
Gives an overview of the Scalation simulation DSL, implemented in Scala. Discusses Scalation motivations and realizations, and how they make it a good fit for Scala. Mentions use of Actors for parallel simulation, but does not discuss further. System built on event graphs, "where the nodes represent types of events and the edges represent causal links between the events."

- https://github.com/scalation/scalation
- Scala Parser Combinator Library.
- What is the difference between a DSL and a library/ package?
- Hyrid Functional Petri Nets.
- Fortress.

Chapter 6

Software

6.1 Spreadsheets

Calculation View: multiple-representation editing in spreadsheets [11].
Presents "Calculation View" (CV), a novel alternate view in Microsoft Excel. Allows users to edit spreadsheets in a more high-level way. Introduces ranges, named ranges, pseudocells, and a block detection algorithm. Details further work to expand CV view, including text interface and expanded features.

- Can CV store intermediate values that the user might not need represented on the sheet?
- Can CV be used on its own? Is there a use case for CV being used on its own?

Elastic Sheet-Defined Functions: Generalising Spreadsheet Functions to Variable-Size Input Arrays [9].
Defines formal syntax and semantics for generalising spreadsheet functions to variable-size input arrays (ranges). Outlines algorithm for identifying generalizations and interpreting most likely generalizations given multiple options.

Chapter 7

Systems History

7.1 Template to Copy

Summary

Context

Discussion Points

Significance

7.2 The Education of a Computer [6]

Summary This paper goes through iterations of "educating" a computer, to where the mathematician using a computer gets the boot, the computer becomes the mathematician, and the programmer becomes an integral part of the computer. A subroutine is a function-like that performs some computation. It has an entry line, exit line, result line, argument lines, and routine lines. Different procedures interact with different lines of the subroutines. The computer is given a bunch of smaller mathematical subroutines that the programmer can use to help construct their programs. It is hypothesised that more "subroutines" could be developed and combined. It is unclear to me what the correlation is between subroutines and modern day functions.

Context I think this work is difficult for modern day programmers to understand/ fathom (myself included). It is difficult to think of what may lie between assembly-style jumps and Haskell, for example, which is the space that this paper explores.

Discussion Points (1) Page 245 discusses turning using programs that contain subroutines as a subroutines itself, which I would consider a profound observation. Yet the "conclusion" focuses just on the arithmetic type mathematical advanecs. Hmm. (2) The proposed UNIVAC is claimed to "not forget" and "not make mistakes." We all know that that is not the case.

7.3 The FORTRAN Automatic Coding System [1]

Summary The paper proposes FORTRAN, a language resulting from a need to reduce computing overhead. The authors report that the goal of FORTRAN is for programmers to be able to use a concise language to specify procedures, rather than writing the granual 704 code directly. FORTRAN achieves this by introducing both a high level language a mechanism of automatic translation to produce optimized 704 code. The language presented uses statements that modern day programmers would recognize - functions, loops, print, etc.

The technical contributions are quite great. The FORTRAN language itself introduces new statements that programmers can use to write pseudo 704 code, with each FORTRAN statement translating to 4 to 20 704 statements. This greatly reduces the time and effort required to write programs and debug. The FORTRAN translator automatically translates FORTRAN statements to 704 statements, and while doing so it makes a number of optimizations that allow for efficient 704 code to be produces. This 704 code may even be more efficient than what a programmer may have wrote directly, and the translator is able to detect patterns and optimizations for different high level ideas. The translator also is able to catch a number of errors and help the programmer debug.

Context This paper's goal was to develop a tool to make programming easier, and it does so by developing a high level representation of frequently used code and introducing a method of translating that high level representation to optimized low level code automatically. Really, that goal hasn't changed much since this time. Obviously languages and toolchains have evolved, but most modern languages employ this automatic translation/ optimization process, and for good reason. It is likely that this paper laid a large portion of the foundation for modern languages.

Discussion Points (1) Not a discussion question, but I belive last class it was said that FORTRAN was originally intended for researchers. In retrospect it is easy to see why FORTRAN-like languages would be appealing for most programmers, not just researchers, but I am interested in how this transition arose at the time. (2) Hmm, I actually keep getting kind of confused by the use of "subroutine" in each of these papers, how is a subroutine different than a function? I read the other paper for this week first, and I was mentally translating "subroutine" to "function", but this paper makes a distinction. Can we clarify the difference?

Significance This paper was definitely significant, as it likely paved the way for most modern languages. Not only does it introduce two major components, the lanaguage representation and the compiler, but it also presents a very compelling case for why each of these is useful. This paper likely introduced a lot of motivation for creating tools that could help programmers be more efficient and more accurate.

7.4 Recursive Functions of Symbolic Expressions and Their Computation by Machine [8]

Summary This paper defines a formalism for defining potentialy recursive functions and proposes S-expressions, S-functions, and S-function apply, which together are a system of theoretical

program representation that is independent from its machine implementation. The paper then discusses the representation of S-expressions as lists in IBM 704 memory. This work addressed a need to represent LISP symbolically, and it offered additional tools to be used in mathematical logic and theorem proving.

The key ideas include: S-expressions, symbolic expressions composed of ".", "(", ")", and an infinite set of atomic symbols. Operations upon S-expressions. M-expressions, or functions of S-expressions. S-functions, which are described by M-expressions, and the S-function apply. The implementation of S-expressions in 704 memory as lists, where the list structure is specifically mechanized to implement an early iteration of garbage collection.

Context This paper discusses LISP for the IBM 704. Technologies have changed dramatically, and we no longer use the IBM 704. LISP is still used, and I'm sure its use cases have expanded past those listed in Section 4. S-expressions are still frequently discussed, although I think because of their simplicity they are most discussed in a pedagogical context. I ran into them when I was googling about Rust macros.

Discussion Points (1) The description of the formalism and the methods reminded me of "An Introduction to the Theory of Lists" by Richard S. Bird, in the sense of, they both describe similar operations, one upon S-expressions and the other upon lists. Any relationship there, or no? This potential correlation seems most relavent to the section that discusses representing S-expressions as lists. (2) It seems like the authors of this paper and the FORTRAN paper both naturally extended functions to include higher order functions. Maybe this is just me not understanding things, but I would have expected higher order functions to be some crazy difficult problem, but I guess that is not the case. (3) Section 4 lists that LISP was used to write the compiler to compile LISP programs. I believe this is called "bootstrapping"? Is this the first instance of this? / I forget what the FORTRAN compiler is written in.

Significance This paper primarily proposes a theoretical program representation and secondarily presents the machine implementation. This is in contrast to the FORTRAN paper, which integrated the "theoretical" language design into the physical implementation, and likely had significant impact on mathematical foundations of computer science. This paper also likely laid foundations for garbage collection.

Personal Assessment

7.5 Garbage Collection in an Uncooperative Environment [3]

Summary This paper discusses a method of garbage collection that does not require any cooperation from the object code generated from the source program. This means that garbage collection can be isolated - allowing for simplier implementations in conventional compilers and for garbage collection overhead to also be isolated such that programs that do not use garbage collection no longer suffer from this incurred overhead.

The garbage collection algorithm presented is the mark-sweep algorithm, which employs two passes over the data to perform collection while gaurenteeing that accessible data is never

corrupted. The first pass "marks" all of the data that is accessible by the program and the second pass returns inaccessible objects to a list of free memory.

The authors discuss several issues that arise when thinkng about garbage collection, but the most notable of their observations is that the problem of whether or not a particular item is accessed in any possible iteration of the program is not decidable.

Context Well, garbage collection is still used today, and it will continue to be used for the forseeable future. I don't know enough about modern garbage collection to synthesize how this paper may have influenced modern designs, but the problems discussed in the paper are still problems faced today.

Discussion Points Last class we discussed NVM (spelling?), which was something along the lines of "persistent memory". How does garbage collection work in this environment?

7.6 Bringing the Web Up to Speed with WebAssembly [5]

Summary This paper introduces WebAssembly, a portable low-level "language" (bytecode) intended to be used across most modern web browsers. WebAssembly is designed to be language, hardware, and platform independent, and it is abstract over programming models. The authors introduce WebAssembly semantics, memory access, control flow, and type system. They discuss browser implementations and performance results. WebAssembly is slower than native code but faster than asm.js, but it also is smaller in resulting size.

The control flow and type system are two of the main technical contributions from this paper. WebAssembly uses a structured control flow, where the design bakes in the gaurentee that the code does not contain irreducible loops or "bad" branches. Control flow elements monitor their own local operand stacks, and branching in the code clears an operand stack, such that users don't have to track information about the stack. This design allows the type system to validate the code in a single pass.

The WebAssembly type system is interesting. It is written in such a way that all of the reduction rules are sound. This means that type-correct WebAssembly code contains so invalid calls, illegeal accesses, and is memory safe. The type system is inherently connected to the control flow design - the control flow design allows the type system to validate the code in one pass and the type system implies that the layout of the operand stack is determined statically. This establishes memory and state encapsulation.

Context WebAssembly is recent, this paper is just from 2017. I'm glad we read this, I see lots of people online really excited about the implications of WebAssembly. The control flow design is novel, although the idea of baking into the type system some desired behavior is not novel.

Discussion Points 1) Why is JavaScript the only natively supported language on the Web "by historical accident"? 2) What would be the closest relatives to WebAssembly's type system? 3) It seems like the most recent languages to be put into the spotlight (Rust, WebAssembly) have core features baked into the type systems so that their target behaviors can be statically determined. Should we expect this to be an upcoming trend or is this coincidence?

Significance WebAssembly is very significant, and much needed. It opens up many possibilities for what one might be able to do on the web, and it ensures that those new possibilities are done relatively more safely. I am excited to see more developments in WebAssembly.

Bibliography

[1] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, L Mitchell Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

[2] Richard S Bird. An introduction to the theory of lists. In *Logic of programming and calculi of discrete design*, pages 5–42. Springer, 1987.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[4] Tongfei Chen. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 45–50, 2017.

[5] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[6] Grace Murray Hopper. The education of a computer. In *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, pages 243–249. ACM, 1952.

[7] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[8] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. 1959.

[9] Matt McCutchen, Judith Borghouts, Andy Gordon, Simon Peyton Jones, and Advait Sarkar. Elastic sheet-defined functions: Generalising spreadsheet functions to variable-size input arrays. November 2018.

[10] John A Miller, Jun Han, and Maria Hybinette. Using domain specific language for modeling and simulation: Scalation as a case study. In *Simulation Conference (WSC), Proceedings of the 2010 Winter*, pages 741–752. IEEE, 2010.

[11] Advait Sarkar, Andrew D Gordon, Simon Peyton Jones, and Neil Toronto. Calculation view: multiple-representation editing in spreadsheets. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 85–93. IEEE, 2018.

[12] Aaron B Todd, Amara K Keller, Mark C Lewis, and Martin G Kelly. Multi-agent system simulation in scala: An evaluation of actors for parallel simulation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011.

[13] Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.