Monetary Economics Simulation: Stock-Flow Consistent Invariance, Monadic Style

Pierre Boudes, Antoine Kaszczyc, and Luc Pellissier

{boudes, kaszczyc, pellissier}@lipn.univ-paris13.fr
Université Paris 13, Laboratoire d'Informatique de Paris-Nord,
93430 Villetaneuse – France

Abstract. An agent-based simulation of a monetary economy as a whole should be stock-flow consistent [7]. We aim at providing a compile-time verification of the preservation of this invariant by the computation. We guarantee this invariant by wrapping the accounting operations in a monad. Our objective is to increase the confidence in the SFCness of an existing complex simulation with a minimal refactoring of code.

Keywords: Stock-Flow Consistency, Agent-based simulation, Functional Programming, Monads, Static Typing, Strong Type Discipline.

1 Introduction

Computer-based modelisation and simulation in economics have taken an important and increasing place. First as a tool to solve differential equations, then, with the advent of agent-based models, as a powerful way to devise and study systems not necessarily near the equilibrium. Computer programs are hand-made and checked by their creators or by third-parties to ensure they satisfy certain properties. However, this has been deemed insufficient in so-called "critical systems", such as medical or aeronautic systems. As the desire for security (both in terms of ensuring the program does what it is meant to do and in terms of not jeopardizing the users' lives) increases, methods to ensure correct-by-construction programs got more popular[1]. As it turns out, even industrial-strenght programs built by experts and used worldwide can still contain a flurry of bugs [17], sometimes critical (for instance, the Therac 25 radiation-therapy machine caused in the mid 1980s the death of at least four people, see [10]).

More testing and more careful thinking in the writing of the programs is an important way to get rid of these bugs, but are both expensive and incomplete. Formal methods can be used and present impressive results, but can be difficult to set-up. They moreover require to have a formal idea of what properties the program should have, which is not always clear (for instance, no specifications are known for fundamental operations in programming language design, such as parsing). What we will advocate here is more of a small change in programming

[1] For example the A340 and A380 fly-by-wire systems were proven free of any run-time errors

style than a change in tools or design. A change in style that allows to enforce certain invariants during the computation, and to verify this enforcement statically, that is, before even running the program.

A natural invariant that can be imposed on economics simulations is the Stock-Flow Consistency (SFC), that is, a model with no unbalanced monetary creation allowed: every new credit is simultaneously balanced by a debit of the same value.

We will capture the SFC condition as a monad. A monad is an abstract and general mathematical concept, infusing in all mathematics and theoretical computer science. We will only present it in a programmer's perspective, but a precise account of its usefulness and ubiquity can be found, see e.g. [16,13]. In a sense, a monad encapsulates a certain notion of computation (use of inputs and outputs, global and permanent storage, data flow exception mechanism, effects, *etc.*) while still allowing to retain all the benefits of a strong static type discipline.

In this article, we first recall the advantages of programming with a functional style and a strong static discipline. Next we give a brief introduction to monads as a way to regain features of general purpose programming languages while staying inside the secure world of pure functional programming. We then apply the monadic style programming to Jamel, a monetary economics simulator, to enforce the stock-flow consistency. After discussing compositions of monads, we will turn back to presenting our implementation.

2 The Functional Style

Functional programming is a paradigm of programmation that "takes functions seriously". This means that functions are first-class objects, in the same way that, say, integers, and can be passed as arguments to other functions. This is reminiscent of traditional mathematics, where it is customary to write what some call *functionals* (functions that take functions as argument), such as the integral over a specified interval.

Another feature of mathematics is that variables are defined once and for all: one never writes an equation like $x = x + 1$, where the intended meaning would be to redefine x. Similarly, if x is a variable bound to a list, a function call $f(x)$ should not change the value of the list. This persistence of data is usually achieved by disallowing mutation in functional programming. *Referential transparency* is another important property, introduced by Strachey in 1967 [15], related to persistence and enforced by purely functional programming. It allows to substitute an expression for another expression with the same value in a computation. For instance, if $f(x)$ is used to compute a new list y, then once that new list has been computed, let say by evaluating the instruction $y = f(x)$, we can indifferently use the already computed value y or compute it again by using the expression $f(x)$ instead of y in the program. The only observable effect of the substitution would be on the execution time.

Apart from an ease of reasoning, immutability also offer great practical advantages. For instance, immutable data structure are not subject to data race

conditions, which allows for safe multithreading. In general, all side effects are avoided, that is, in a pure functional programming language, we know that a function of type `int list -> int` takes a list of integers as an argument, returns eventually an integer, but never changes any global variable, prints something,... In particular, the execution of a pure functional program does not depend on the state of the environment. It is always the same, which allows for easier debugging.

The type of a program is an abstraction of its behavior. Since we might want to have behaviors with side-effects, two solutions are possible: either relaxing the conditions, allowing for unpure programs with seemingly pure types, or enriching the type system. This is the road we will follow.

3 The Monadic Style

Consider the running example of agents that have an internal state `state` (that may be composed of their happiness, their hungriness, their tiredness,...). Each agent can trigger an action, that acts on their state (the actions are thus functions `state -> state`. The actions compose. As such, an agent in state `s` successively eating, reading and sleeping would be represented by:

```
sleep (read (eat s))
```

Things get more complicated if the actions require to do something else than just modifying the state of the agent. For instance, consider that the agent now wants to keep a log of what they have read (notes, for instance). The type of the function read would become

```
read : state -> (state * notes)
```

but in this case, the function read does not compose with itself anymore. That is, two sessions of reading cannot be represented by `read (read s)` as the inner parenthesis is not of the right type. The solution comes by thinking on what should happen with the notes of the first reading session when the second begins. They should not be discarded, but, on the contrary, the notes should be appended. So, the read function should take as an extra parameter the notes already collected, and return not only the state but the new notes. Alternatively, for reasons that will become clear, we will present it as a function that takes the state, and returns another function that takes the notes and return both the new state and the new notes appended to the old ones[2]. As such `read` gets typed by:

```
read : state -> Bookkeeper(state)
```

where we write the type `(notes -> state * notes)` as `Bookkeeper(state)`.

We now need to think on how to compose functions that have such a baroque type. Consider, more generally, two functions

[2] This operation, called curryfication, is benign from a logical point of view

```
f : 'a -> Bookkeeper('b)
g : 'b -> Bookkeeper('c)
```

where `'a`, `'b` and `'c` are arbitrary types. In order to compose them, we need a mechanism to lift `g` to a function of type `Bookkeeper('b) -> Bookkeeper('c)`. The function:

```
fun (book: Bookkeeper('b)) -> fun (n:notes) ->
        let (b, n') = book n in
                g b n'
```

takes two arguments: first a function of type `Bookkeeper('b) = (notes -> 'b * notes))` and second an element of type notes. The function waits for a note to append it to its own notes and return its result and we can compose this function with `f`.

In the same way, the eat function might be impure, the side effects being the money spent in the process of buying food. The interactions with the financial system may be encapsulated and `eat` be typed as

```
eat :  state -> Finance(state)
```

where `Finance('a)` is defined as `accounts -> ('a * accounts)`. The exact same trick allows to compose two functions of this type. This is a really generic and pervasive structure across all mathematics and computer science: the monads.

3.1 Formal Definition

A monad is a construction on a type such that:

- for every type `'a`, there is a type `M'a`
- for every function `f:'a -> 'b`, there is a function `Mf : M'a -> M'b`
- for every type `'a`, there is a function `return : 'a -> M'a`
- for every types `'a` and `'b`, there is a function `bind : M'a -> ('a -> M'b) -> M'b`.[3]

which satisfies the following:

- for every types `'a` and `'b`, every function `f:'a -> M'b`, and every `a:'a`, `f a` = `bind (return a) f`
- for every type `'a` and every `ma:M'a`, the following equality is true : `bind ma return` = `ma`
- for every types `'a, 'b, 'c`, every `a:'a`, every `ma:M'a`, every function `f:'a -> M'b`, and every function `g:'b -> M'c`, the following equality is true : `bind ma (function a -> bind (f a) g)` = `bind (bind ma f) g`

[3] This is the traditional presentation in the programming languages community. It may be clearer to reverse the order of the arguments and present it as a function of type `('a -> M'b) -> (M'a -> M'b)` that lifts a function

Intuitively, a monad encapsulates a principle of computation, for instance, in the two examples above, the Bookkeeper monad encapsulates the ability to log information and the Finance monad encapsulates the ability to access and modify an account. The other axioms are just here to ensure that composition is always possible, and that a program that doesn't use a particular principle of computation can always been seen as using this principle (even in a trivial way).

4 The `Financial` Monad

The `Financial` we propose resembles the `Bookkeeper` monad. It is defined as follows.

- For any type `'a`, `Financial 'a` is equal to `Accounts -> 'a * (Order list)` where `Accounts` and `Order` are two types. `Accounts` is the type of a map of keys-values where each key, let say of a type `Id`, identifies an account and its owner (each owner may have multiple accounts) and the value, let say of a type `Amount` is the balance of the account all expressed in the same currency. Lets think of this as an array or a list for the moment. The `Order` type contains at least a triple `Id * Id * Amount` And `Accounts` shall implement a method `execute` which takes an order and transfer the amount from the first to the second account. An account may be subject to the restriction that its balance is sufficient to perform the order (of course, there is at least one account, owned by the central bank, which is not subject to this restriction for the sake of money creation). So orders may fail silently. To stick to a functional programming style, and to simplify the presentation one can suppose that lists and maps are immutable and that the `execute` method gives back a new map where the accounts have been updated. In a real implementation we can use caching and memoization to preserve this immutability abstraction without sacrifying the performance.
- For any type `'a`, and any `a` of type `'a`, `return a` gives back `a` and the empty list : `return a = (a, [])`
- For any types `'a` and `'b`, any `ma` of type `M'a` and `f` of type `'a -> M'b` :

```
bind ma f = fun (accs1: Accounts) ->
 // Get the state and previous orders from ma
 let (a, orders1) = ma accs1 in
 // Apply previous orders to accounts to obtain new accounts
 let accs2 = List.fold_left
   (fun accs2 -> fun o -> execute accs2 o) accs1 orders1
 in
 // Get the new state and the new orders from f
 let (b, orders2) = f a accs2 in
 // Return the new state and all orders (previous and new)
 (b, concat [orders1, orders2])
```

The important point here is that `b` is computed from the accounts after all the orders put by `ma` have been executed.

One easily checks that this triple satisfies the monad laws.

In a simulation using that monad, every agent who usually performs money transactions will produce a function of type `'a -> Financial 'b`, for a certain type `'a` and a certain type `'b` and all these functions will be composed using `bind`. At any moment one can (re)compute the actual values of the accounts using the list of orders carried by the last monadic value.

We may have defined this monad by taking

```
M'a = Accounts -> 'a * Accounts
```

but this won't have garantee the stock-flow consistency.

With a reasonable type-checker like the ones of OCaml or Scala, even if some part of the code is using type casting (`magic` or `asInstanceOf`) during the computation if we get a value truly of type `Financial'a`, then we easily guarantee by construction that a value of type `Financial'a` involves a computation which cannot violates stock-flow consistency, which is not the case with `M'a`.

4.1 Implementation and benchmarks

We made an implementation of our *Financial* monad. In this section we explain how we managed to make it work with *Jamel*, a monetary economics model, and how Jamel took advantage of it. Let us start with a presentation of Jamel: it is "a macro-economics agent-based model with heterogeneous households and firms, and a banking sector. It provides a decentralized and stock-flow consistent framework for the simulation and the analysis of economics dynamics". It is developped by P. Seppecher and I. Salle (universities of Paris Nord, Nice Sophia Antipolis, and Amsterdam).

The model defines three types of agents:

- *Banks* (there is only one)
- *Firms*
- *Households*

Each of these agent types have a defined behavior:

- The bank lends money, earns interests for loans, and shares dividends with his owner (some household).
- Firms hire workforce, pay wages, sell goods, borrow and pay back money and shares dividends with his owners (some households)
- Households earn wages, work for firms and buy goods.

The initial number of agents is defined in the *scenario*. The scenario holds the exogenous arguments of the simulation, for example:

- the number of households and the number of firms

– the duration of loans, the propensity to save money for households
– a production shock at a certain time.

The simulation takes places in sequential periods, and a period takes places in the following sequential phases (non exhaustive):

1. The Bank shares dividends
2. Firms share dividends
3. Firms plan production
4. Firms hire households in the labor market
5. If needed, firms borrow money from the Bank
6. Firms pay the wages
7. Firms produce
8. Households consummate
9. The Bank recovers loans by the due date

Households never borrow money, this is the reason why wages are paid before production takes place.

Note that between each period can remain some money (hold by firms or households) and some goods (unfinished or unsolded). Note that the money is endogenous and is created by the bank when it lends money. Note that Jamel is determinist, and two identical scenarios lead to two identical simulations.

Now let us study the banking system at a closer look since this is the main point when we are studying SFCness. The bank holds an `Account list`. But theses `Accounts` are different than those presented in the Financial monad section, so let us call them `JamelAccounts`. Each firm and each household have his own `JamelAccount`. Each `JamelAccount` has a `debt` value and holds a `Deposit` with an `amount` value and methods for `credit` and `debit`. The bank has no `JamelAccount` for itself, but it holds three values: `liabilities`, `assets`, and `capital`. Also, there are two additional mechanisms:

Loans the Bank lends money to firms, which must pay interests and pay back some day;

Cheques some agent promise some amount to another agent, which trigger the debit attempt, and if it is successful, trigger the credit of his own deposit.

Let us look at an example. What happens when "Firm A" wants a `Loan` of "100" from the bank ?

1. `Debt` of the `JamelAccount` of "Firm A" is increased by "100"
2. `Assets` of the bank are increased by "100"
3. `Capital` of the bank is increased by "100"
4. `Credit` method of the `Deposit` of "Firm A" is called with "100", which means:
   (a) `Liabilities` of the bank are decreased by "100"
   (b) `Capital` of the bank is decreased by "100"
   (c) `Amount` of the `Deposit` of "Firm A" is increased by "100"

Observe that the SFCness is separated in different values and kept in different structures (also shared with SFC unrelated values). How do we express the same story with our Financial monad ?

1. `Transaction from ‘‘Bank Assets’’ of ‘‘100’’ to ‘‘Firm A’’`

This is an `Order` meant to be applied to an `Accounts array`, as explained in the Financial monad section. The order of the example will results in a decrease of “100” in the “Bank Assets” row and an increase of “100” in the “Firm A” row of the `Accounts array`. Note that we created two `Ids` for the bank: “Bank Assets” and “Bank Capital”.

Now you understand the advantage of using the monad. Jamel promises SFCness by occasionally call a method wich sums the amount of `JamelAccounts` and compare it with the bank values, *etc.*. The Financial monad uses the following *atomic* orders:

Creation order give it some *non already existing* `Id` to be created;
Transaction order give it two *existing* `Id` and an `Amount` to transfer between the two);
Destruction order give it an `Id` to destroy and an *existing* `Id` to inherit the `Amount`.

By atomic we mean that these orders are executed in only one step. One `Order` must satisfy the rules defined above. If it does not, the `Order` will be not applied by the `Accounts array` because it would break SFCness. For example, if you destroy an `Account` with some remaining money, global system loses this amount of money.

Not everything is so easy, there is a gap between the program of the monad and the program of Jamel. We take advantage of the good understanding between *Java* language (of Jamel) and *Scala* language (of Financial). But it is not enough. To statically assure SFCness, we would have to use the monad everytime a money related action is done in Jamel. Due to the *immutable* nature of the monad and the *mutable* nature of Jamel, this would be not possible without a lot of changes in the code of Jamel. We want to change the least, so we make a compromise. We keep the `JamelAccounts`. During each period, we collect `Orders`, by translating Jamel's monetary actions. At the end of each period, we submit them to the Financial monad. The we compare the `Accounts array` of the monad with `JamelAccounts` and the values of the Jamel bank. If there is some differences, we stop the simulation. This give us two things:

- an `Order` list obtained from Financial which is statically SFC;
- `JamelAccounts` which can perform non-SFC actions during runtime, but will cause the simulation to stop if it do so.

Considering the minor changes needed to Jamel's code, we think this is a acceptable patch and an reliable static check of the SFC of the simulation.

5 Composing with the `Bookkeeper` Monad

The fact that every agent accounts well has no impact on the global consistence of the financial system. So, the SFC condition can be factored in two parts: a global

part, the aforementioned `Financial` monad, that ensures that the monetary creation happens in a controlled way ; and a local part, that ensures that every agent does its accounting correctly.

This is a fairly simple monad, essentially a variant of the already-presented `Bookkeeper` monad, that logs independently for each agent its action, in a double-accounting style. Supposing we have chosen a data-structure to represent the accounting (a list of vectors, a bi-dimensional table,. . .), it only appends the new information to this structure.

5.1 Implementation and benchmarks

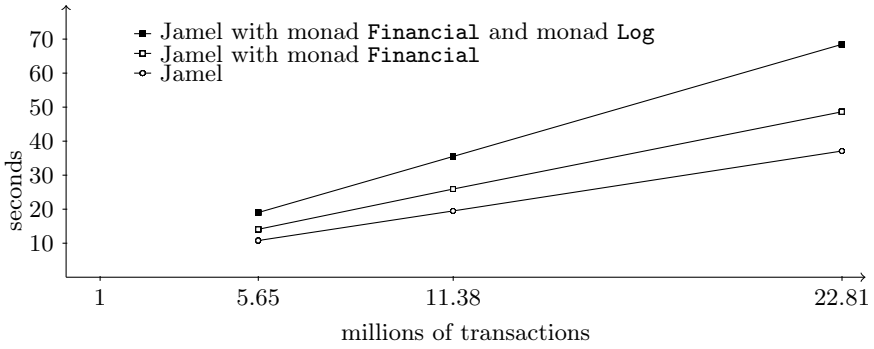

Fig. 1. Average execution time

For our implementation we created a *Log* monad which is just another name for the Bookkeeper monad. Our Log monad keeps trace of operations, under a chosen format. We chose the *RDF* format, which is structured as a *Triple* : *Object Predicate Subject*. For example, an `Order` like `Transaction from ‘‘Firm-A’’ of ‘‘100’’ to ‘‘Bank-Capital’’` will result in the following RDF triples :

- `pref:transaction-X-Y pref:hasForTransmitter pref:Firm-A`
- `pref:transaction-X-Y pref:hasForAmout ‘‘100’’`
- `pref:transaction-X-Y pref:hasForReceiver pref:Bank-Capital`

On can set the prefix `pref` as necessary, for example an identity alias of this particular execution of the simulation. X is the period number, and Y is the order number of this period. Observe that RDF triples can be seen as a graph. We aim to study this particular data structure and hope to extract some semantics. So our Log monad translates operations and carry triples during the simulation. In the background, we are using a *RDF store* to delegate the task of handling a potentially huge amount of RDF triples. This is just a mechanism and in the foreground, our Log monad present the same type as Bookkeeper. But we are tied to use it that way `Financial (Log 'a)` and not `Log (Financial 'a)`,

because we are generating the RDF triples from happened operations, and the other way makes little sense.

In Figure 1 you can see benchmarks of Jamel and our monads. There is a cost to use the monad because monetary actions are obviously the most used objects in the simulation, and use them twice cannot be free. The good point is that the additional time is reasonable.

6 Conclusion

Types systems are powerful abstractions allowing to reason about programs. What we presented is a first attempt at allowing the type system to know something about the nature of financial economies modeling. Encapsulating all the monetary interactions inside a monad allows to know that well-typed agents cannot 'interact wrong' [4]. The benefits are not only practical (allowing a programmer to code sloppily while ensuring certain properties): a type system is a mathematical structure that have been extensively studied under the name of *category* [5]. So a particular type system (such as for instance, a type system enriched with a `Financial` monad) is a category with some specified extra structure. Finding an appropriate type system for monetary simulation provides a path toward an alternative understanding of the mathematization of the economy.

A work that have been a great motivation for us is [2]. A type system for classical mechanics is given, allowing to write programs that describes systems in terms of their Lagrangian. In this framework, a well-typed program is ensured to satisfy, for instance, the conservation of energy or of momentum. One can say that this bridges the gap between the theories (texts and equations on a textbook) and the simulations (computations on a computer) by deeply embedding the theories inside the computing environment.

Adapting an existing simulation involves a rewriting of the codebase. The amount of code involved in such a rewriting depends on the style initially used in the simulation. The easiest situation arise when the code updating attributes of the debtor of a money flow and the code for the creditor live in the same method call. Mainly this a situation where the method implements the interaction of the two agents and its effects at both sides. Other situations arise, with nested method calls and involve more rewriting to put side by side all the financial transactions generated by a single interaction.

Interaction-oriented agent-based models [9] may prove particularly well-suited for internalizing invariants in the monadic style. This is in line with a much broader view that computing is not about states of objects or agents but is centered on their interaction [4]. Interaction has been a subject of great interest in

[4] To shamelessly and agrammatically adapt Robin Milner's oft-cited "well-typed programs cannot 'go wrong'" [12]

[5] This is the third leg of the Curry-Howard correspondance, the dictionary at the heart of modern logics and informatics

theoretical computer science. Actually, a whole framework, the *π-calculus* designed to reason about concurrency (and so, the problems of communication, be it synchronous or asynchronous, between agents) has found applications in varied computing areas [6]. A seemingly unrelated domain, proof theory, has been infected in recent years with research programs with name witnessing their interactive origin, such as *game semantics* [8] or *Geometry of Interaction* [6].

Interactions *per se* have also been an object of study in theoretical economy, as witnessed by the game theory pioneered by [14]. In a game, two or more players make moves, so it seems natural to model a game as trees, each edge representing a position, and each vertex the accessibility relation (the fact that a position can be accessed from another). Such abstract games enjoy a close relationship with categories with structure, as exemplified in [11]. So, in a sense yet to be made precise, approaching a program via the a search of its most suitable type system may offer naturally an interpretation of the program in game-theoretic terms, and conversely, to ground firmly such games in a setting both algebraic and algorithmic.

It is our belief that insightful models of our world have a triple nature: analytic, algebraic and algorithmic. For instance, the Dirac equation, describing the electron in a relativistic setting can be seen either as an equation (analytic facet), as its symmetry group (algebraic facet)[7], or as a discrete computation process (algorithmic facet)[8]. We believe that type systems (algebraical abstraction of programs) are instrumental in achieving that.

References

1. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115, 2001.
2. Robert Atkey. From parametricity to conservation laws, via noether's theorem. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 491–502. ACM, 2014.
3. Davide Chiarugi, Pierpaolo Degano, and Roberto Marangoni. A computational approach to the functional screening of genomes. *PLoS Computational Biology*, 3(9), 2007.
4. Pierre-Louis Curien. Symmetry and interactivity in programming. *CoRR*, abs/cs/0501034, 2005.
5. C.A. Dartora and G.G. Cabrera. The dirac equation in six-dimensional so(3,3) symmetry group and a non-chiral "electroweak" theory. *International Journal of Theoretical Physics*, 49(1):51–61, 2010.
6. Jean-Yves Girard. Towards a Geometry of Interaction. *Contemporary Mathematics*, 92:69–108, 1989.

[6] Let us cite [1], that applies the π-calculus to cryptography, or [3], that applies it to computational biology, among many other cases

[7] see e.g. [5]

[8] see unpublished work by Pablo Arrighi

7. W. Godley and M. Lavoie. *Monetary Economics: An Integrated Approach to Credit, Money, Income, Production and Wealth.* Palgrave Macmillan, 2007.
8. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.
9. Yoann Kubera, Philippe Mathieu, and Sébastien Picault. IODA: an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3):303–343, 2011.
10. Nancy G. Levinson and Clark S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, July 1993.
11. Paul-André Melliès. Game semantics in string diagrams. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 481–490, 2012.
12. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
13. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
14. John Nash. Non-cooperative Games. *The Annals of Mathematics*, 54(2):286–295, 1951.
15. Christopher Strachey. Fundamental concepts in programming languages (1967). *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.
16. Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149 – 168, 1972.
17. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference PLDI*, pages 283–294. ACM, 2011.