ProjectR functional-prototyping report

Christian Weilbach

25. August 2013

Contents

1 Introduction

1.1 Motivation

Functional programming is currently living through a kind of silent renaissance after it had been dragged into the AI winter at the end of the 80s together with the language Lisp and has not seriously affected mainstream programming, but only in spots. There are concrete reasons for the failure of Lisp [26] [18], although the avoidance of functional abstractions compared to object oriented abstractions is still mysterious to the community of functional programmers at large. We will try to understand the historical development in terms of the software industry and its theory of computation to shed some light on how to approach the design decision for a functional environment today, given the obstacles. We assume that the reader is familiar with some imperative programming as well as aware of the informal vocabulary as is used just to name two popular examples, in Java or Python. Not all terms are defined, only ones that seem critical for the following argument. The reader can faithfully skip functional terms and expect them to become clearer through the article.

No functional programming skills are assumed, although functional programming will still not be a easily familar but an abstract path with new insights in the simplest abstractions on different levels in different contexts, so additional recherche for the reader is continuously reasonable and theoretical abstractions can prove mind-twisting, obscure or confusing in the beginning (as is the case with imperative programming, too). We have tried to cite many online resources for that reason. To properly digest we recommend a relaxed playful approach to the abstractions [22][24] and don't introduce them unless we can and do give strong practical reasons.

Since we want to evaluate functional programming for game develop-

ment specifically and practically here, we have to ask why the software-gaming-industry is barely interested in functional programming and why we should derive. We have few information about the actual private design decisions of game publishers as the industry is both fast paced and highly competitive, but we can for now assume that the reasons are similar to the software-industry in general, possibly emphasized by the high turnover speed and throw-away products with short maintenance cycles. This is the case because game-programming converges and overlaps in many dimensions with traditional software engineering and can be seen as a special case of it. This is especially the case for the game concept at hand, as it merges reality with the game world, it is an augmented reality simulation. In fact it is taking traditional massive multiplayer games and current augmented reality prospects to the extreme when it tries to model parts of reality as a game world.

So reframing the specific design problem more generally is asking why the industry sticks to imperative programming and how functional programming can be reasonable, demonstrated on an augmented reality gaming simulation representing the current trends of distributed high-availability systems on the web and in mobile apps. Many reasons against the need for functional programming have been put forward repeatedly, we pick three covering most criticisms:

1. the *lambda-calculus* is not more than *turing-complete*, all problems solved in functional programming can also be solved by imperative programs

2. functional programming abstractions add overhead and make programs less efficient, prohibiting low-level optimization shortcuts in cases of bottlenecks

3. functional programming is an invention of academics and mostly solving problems that have been theoretically made-up before.

The first point is most critical as it points out the common wisdom of every programmer that one should not build in stuff that is not related just because it might theoretically become useful (later). Yet this only begs the question which abstractions to chose in the first place and why as well as what programs need to be adaptable. We will see later how this argument misperceives expressiveness of a model of computation with its computability.[27]

It is also of course an additional burden on the already stressed programmer. To approach the problem properly we have to start with an assessment of problems in imperative programming and then judge whether functional programming solves them as this is the only way that abstractions make sense for practical programming (running processes).

We therefore have to first elaborate how imperative programming is done today and thoroughly study its weaknesses on its history and some examples and then reapply the lessons learned to our game prototype.

1.2 Imperative Programming (place-oriented programming)

Imperative programming always was at the core of modern computers. Starting with assemblers and rising to higher-level languages like Fortran, Cobol and C, most programming tasks until today have been solved using imperative programming environments and continue to do so. Originally programming in assembly was a mere extension of placing different connections or describing punch-card systems with loadable software in terms of memory itself (modelled by von-Neumann-architecture). Programs hence became data in memory and could be manipulated during execution by the program itself.

Programming in processor assembly as well as early languages like Fortran and C have also inherited the traditional mechanical approach of the process by telling the machine in each step how to adjust its connections and what data to load and when often embedding native assembly code in some places. In a time when memory was the most precious part of the machine, each access had to be carefully considered. Memory until the 70s was counted in kilobytes, most early programmers were technical specialists and had to know all memory addresses including system and hardware-specific addresses of their machine and program. Programming was very directly targeted at a single peace of metal and only C and Unix have established platform independent programming in the software-industry targetting main-frames and workstations during the 80s.

The PC industry has still counted memory in very small chunks during the 80s and has for a long time during DOS or Mac OS times allowed direct memory access, because memory was small and system functions could be directly steered by manipulating memory (e.g. using interrupt hooks). DOS also put the programmer in exclusive control of the whole world, as the program was practically running like an operating system and as the

only process.

This mode of programming which is still closely related to the concrete hardware is modelled theoretically as a *register- or stack-machine*. Programs change their environment by sequentially changing and combining memory through its addresses until the result is achieved. We call this result the *state* of the program from now on. The program can be understood as a sequence of states that way, whereby the programmer combines places (memory locations). Usually places are expressed as variables in form of symbols in imperative languages like C and express references to memory (places), not values themselves. Java compared to C++ has started to model primitives like *Strings* and *Dates* as immutable values, but in general state is accessed through references on mutable memory.

Imperative programming hence stems from a hardware-related perspective and has naturally grown everywhere programming has risen, especially in embedded computers targetting domain-specific processes as the way to directly alter a machine by advising the machine through its electronics in each step what to do and which data to use for conditional branching. Imperative programming understood as place-oriented programming is still the predominant mode of programming even in industry-standards like JVM or .Net languages today.

It also continues to be necessary the more constraint the executing machine is, e.g. in embedded chips, although functional abstractions like point-free concatenative programming exist for this space as well[9] and have been applied widely, e.g. in the FreeBSD bootloader. A relevant example for us is that the cheaper but weaker PC hardware has also killed the Lisp-Machine [18] and emphasized the general accelerating economical trend towards cheaper hardware. A similar trend can be seen with ARM-based mobile devices. It is remarkable that this is only a look at the expenses of the hardware not of software, which at the same time rises with complexity of networked applications between small ubiquitous devices.

In general imperative programming has introduced abstractions gradually to place-oriented programming, first by allowing platform-neutral programs being written in Fortran, C or Cobol and compiled to a specific hardware by the compiler targeting many architectures and machines at once, then adopting object-orientation to make systems more extensible and have higher-level abstractions in form of classes with polymorphic-dispatch and finally introducing garbage-collection and managed memory on managed runtimes like the JVM, CLR or runtimes for Python, Perl, Ruby, Scheme

and other modern dynamic languages.

While all runtimes mentioned above have proven maturity now on even very cheap computers like Android-phones < 100 $, memory in most of their supported languages is still written directly by programmers through variable assignments without abstractions. Furthermore this is still considered to be of performance-critical relevance in the same way that garbage-collection is still dismissed by many C and C++ programmers for performance reasons. Yet runtimes like the JVM with garbage-collection have reached roughly a factor 2 of C performance [16] in case of cautious optimizations and the constant memory overhead of the VM has become negligible recently. A majority of the industry has now adopted the notion that managed-runtimes with garbage collection and JIT compilation offset the low-level performance benefits of C and C++ by developper-efficiency and the ability to scale out in many applications, especially on the web.

It is noteworthy for this article that previous dialects of Lisp like *Scheme* and *Common-Lisp* are also programmed in a place-oriented manner, although functional programming with few state changes is already encouraged and supported by abstractions like *closures* and powerful libraries of functions for pure functional-composition, state is modelled by direct assignments of memory.

1.3 Problems with imperative programming

As pointed out in criticism 1. imperative programming can basically emulate any functional behaviour like for instance closures, the core of the lambda-calculus. Functional environments do run on imperatively programmed hardware. Even more, many languages on the newer runtimes like Python, Ruby or some dialects for JVM and CLR support first-class functions, called closures. They have even become part of C++11. They hence also allow to apply the same functional techniques as functional languages have, in fact leaving the choice to the programmer for the problem at hand.

This seems to be much more reasonable and can be understood in the tradition of place-oriented programming with the previous addition of object-oriented programming, most famously in C++ and Java. Scala or F# for instance are attempts to allow selective application of functional techniques in an imperative object-oriented environment. So the complaint becomes even more severe, revealing that functional programming has ac-

tually nothing to add, but only removes "power" in terms of techniques and patterns from the programmer. We can see now why for many imperative programmers the need for a functional environment seems only more like an academic idea (criticism 3.).

The idea of functional programming can hence not be understood simply by prolonging the evolutionary development of toolbelts in form of language-features, combining new ideas with old. Instead we have to study the problems of imperative programming understood in computer-science terms to reason about the abstractions of the lambda calculus.

1.3.1 Formalizing imperative programs

It is a common wisdom with many experienced programmers that programs in fact can be understood and modelled as state machines (a reduced model as finite state automaton is used to parse regular expressions). [1] The "Go"-concept of concurrently communicating through channels is actually implemented as a state-machine which unpredictably executes any of the non-blocked light-processes, called coroutines, and hence can be implemented on any runtime. [5] We have introduced the term *state* above already in an imperative context. The imperative programmer instructs the environment by continuously reading memory, computing a value and writing it to some memory. The common instructions are reading and writing memory by combining variables (references/memory addresses), loop-constructs, conditional branching and a standard library for side-effects (changes to places outside of memory). More complicated features like object-orientation are wrapped around the imperative constructs to compose a whole out of modular units (mostly classes and methods).

If we understand that the imperative programmer changes the state of the whole program logically in every step assigning memory to a new value, we can determine how many states a program has or potentially can have. The complexity of the state-model explodes with each additional assignable variable depending on the range of possible values and branched combinations with other variables in scope. In fact the complexity has no computable upper bound.

1.3.2 Software-engineering experiences with imperative programming

In software-engineering metrics the *cyclomatic number* is used to judge the complexity of code by counting nested loops and composed branches for this reason. Critical studies of the software-crisis, the failure of the majority of software projects to deliver a valuable result in time, have put program-complexity center-stage in attempts to advice programmers in writing cleaner code with less complexity.

Yet all measures and practices only come from the outside to the complexity of the concept embodied in the program and its incidental complexity in form of technical implementation details, often summarized indifferently as boilerplate code. A reason for this is the continuous failure of metrics to generally help improve code-quality except for the rough proporionality of errors to lines of code, emphasizing the quantive aspect summarized as "boilerplate".

Barely do software-engineering studies mention a simplification of the state-model explicitly, although simplification is a common idea. The problem is that software-engineering is removed from the production of the actual program and can for reasons of the halting problem not describe general procedures to fix concrete broken programs. The logically following and currently predominant approach of reducing boilerplate is already intentionally covered by reduced syntax, inference of type-information and focused domain specific libraries which appeal to common patterns and emphasize ease-of-use in e.g. Perl, Python, Ruby or Groovy. As complexity explodes with code-size in imperative programming, inversely code reduction can be an enormous cost reducer.

The failure of the industry has now also for some time been fought by introducing less software-management and closing the gap to the problem space understood through an iterative adaption process. The developers are brought to the client/user through *agile development* processes and break the task into small decoupled pieces. Instead of adding new management guidelines and additional features for complexity reduction to the programming environment (e.g. metrics and static code-checkers into an IDE), removing complexity from development itself has proven healthy and is from the theoretical stand-point of the halting-problem the only way to simplify a (software) process.

1.3.3 Stepping up the game - distributed state and execution

It is also insightful to consider the problems occuring in networked stateful systems, especially since almost all devices will be online in the future if they are not already. Our massive-multiplayer augmented reality concept with mobile clients fits perfectly in this category. While imperatively programming a single program (state-machine) can be easy, it cannot be simply combined with itself or other programs over the network, being it a memory bus between threads or faulty connections over the internet, into a sound state-machine. The complexity with stateful systems explodes, so far that security experts describe the flaws in the millions even for small networks.[14] Although not every bug (faulty state of the machine) gets triggered, understanding how it came to be is almost impossible in such a complex system. This is commonly known in the industry as how hard multithreaded programming (on mutable memory) is. A whole class of problems arise from manual composition alone, being it deadlocks, race conditions or global locks as bottlenecks (e.g. Python's global interpreter lock).

1.3.4 Imperative attempts at solving complexity

Reducing the imperative state-model consciously inside imperative programs is advise given in multithreading literature [11], since locking, the standard approach to synchronize access to places (memory or other side-effects), is very difficult to get right logically and does not compose either. Many imperative programmers try to do so already by consciously dividing memory by different concerns in form of encapsulation and reduction of scope, most notably in object-orientation. Since such references to parts of memory are referencially opaque to the programmer though, state is still as complex, as in fact write-access to memory affects everybody holding a reference to its memory address or some composition of it and is globally leaked because all following states can through branching implicitly depend on the changed memory. Instead of the actual value the program shares information through a memory address (place).

To achieve proper multithreading performance it is today often comparatively cheap to even defensively copy all relevant state in a manual closure and then execute an isolated job with it in a separate environment, e.g. a thread in Java [11], manual coordination between operating-system

processes in Shell or C, Python's multithreading or now more popularly in Javascript with web-workers for instance. Shared memory is deliberately reduced or even avoided.

It is also noteworthy that problems in multithreaded (or in general parallel) programs are often not reproducible and hence cannot be caught by testing, especially the attempt to avoid bugs by the now popular test-driven development (TDD) with automatic unit and integration tests. The imperative model hits a wall here, which can only be avoided by manually doing all the steps of state reduction the lambda-calculus assumes for its basic building-block, the (anonymous) function *lambda*.

To summarize the practical problems and theoretical insights, we can say that the biggest cost and highest risk for software projects today, complexity, can be approached by reducing the states and transitions of the program as much as possible. Since state-changes are embodied by side-effects, most notably access to shared memory, avoiding writes and treating memory as immutable already decouples imperative code, although a single implicitly mutable piece of memory can corrupt all others and stay undetected for long periods of time.

1.3.5 What about Object-Orientation?

The idea of object-orientation as it has manifested in mainstream languages like C++, Java or C# is modelled as an extension of the struct in the C language, being itself modelled after a fixed memory layout with compile-time resolution by memory address (esp. typed method dispatch), confuses many of the underlying problems for its misconception of encapsulation. Instead of really encapsulating state with messaging like the prototype language of object-orientation, Smalltalk, the object shares its notion of time with the system (runtime) although it seemingly lives on its own. The difference can be seen when compared to Erlang's messaging concept, where the system has no state (assignable memory) and each process only uses messages to communicate with others or itself (store state).

In Erlang the state of each process only depends on the immutable messages (values) it has explicitly received or sent, while each mutable object in statically dispatched object-oriented languages can be potentially affected by global state in a non-deterministic way, independent of its inputs, even if only by some other part of the program or some library, in Erlang the very concept of computing (actor-model) outrules mutable state and forces

the programmer to concretize state transitions as immutable values in messages. This underlines why object-relational-mapping (ORM) systems, like Hibernate, Django, Ruby on Rails or Java Enterprise Beans can make no guarantees about state consistency in a parallel (read: distributed) environment, no matter what they claim.

1.4 Functional programming (as value-oriented programming)

1.4.1 Addressing the obstacles and risks for imperative programmers

Functional programming concepts go further down the road of simplification here, trying to model the state-machine of the actual program itself explicitly, not allowing memory assignment freely. The theoretical study and maturity of the lambda calculus and its simple unit of just a function (closure), called *lambda*, allows to model complex functions out of single lambdas which are not able to change their environment (assign memory) and interfere, but rather only return a *value* (not a memory address to a mutable place).

They are like pipes with inputs and output and otherwise completely opaque. Since many functions composed again only resemble a pipe, composition is consistent without additional care. For this reason parallelism has also been a pinnacle of functional programmers since the beginning, as pure computations can be distributed much more easily.

This reduction comes with the theoretical insight that using pure functions means maximum decoupling, as a *lambda* only depends on its parameters, we say it is side-effect-free or pure. It is still having the same computing power as an imperative basis, similar programs can be assembled, having the core logic of the program in pure functions which can be studied, run and combined independently and only explicitly result in a side-effect (changes the state of the program) when the state is consistent and reasonable to the programmer.

It is the crucial point of the complexity-problem to understand that it does not come from concrete technical problems of the programming primitives or environment, but from the inherent logical complexity of programming (sequential logic) itself and the outside complexity of the problem as well as the limited understanding of it by the programmer which makes an approach with few side-effects more reasonable and much more easily adaptable.

It is this self-restraint of making few explicit state transitions that pulls many imperative programmers off when they first face the alien world of the lambda-calculus (with the three criticisms mentioned above). Imperative primitives like loops have the notion of mutability built-in and all abstractions, from functional closures over mutable state to obect-orientation, help the imperative programmer in managing global state through references. Many imperative programmers therefore feel deprived of their way of programming and reduced to mere beginners in programming with only a *lambda* in their hand.

Since most industry-code is written by professionals having little time and interest to investigate solutions not endorsed by their employer, breaking product-lines and basically the whole underlying state-model as well as sceptical colleages (as they have something additional to learn then), adoption mostly depends on enthusiasm for programming or the sheer economic need to get stuff done in the startup scene, special domains of big-data and the web. Industry environments still don't fit the functional attitude well, but changes can be seen.[2][25]

1.5 Using the lambda-calculus

Having framed our approach, we now want to quickly lay out the general functional concept. As Sussman has already mentioned in his lectures about "Structure and Interpretation of Computer Programs"[28], computer science is neither about computers nor really an (empirical) science. It is about modelling processes. We hence then have to ask how to model processes. The lambda-calculus, developped in the 30s by Alonso Church and closely related to the mathematical studies of computers and the attempt to find declarative solutions to programming itself, by letting math solve itself and abandoning the need for manual proof (which every computer-program can be shown to be [7]), is the most mathematical of the equivalent computational calculuses. It covers recursive functions and expresses solutions in terms of functional application which is close to mathematical concepts, but expresses sequential algorithmic behaviour compared to purely declarative math and hence can be used for execution.

We approach our prototype with a dialect of Lisp (LISt Processing) called Clojure. Lisp itself has been theoretically invented in 1958 by McCarthy to tackle AI problems only to be found practically executable in the following years. Lisp uses the minimum syntax necessary to construct a

language for the lambda-calculus, lists. This syntactical homogenity of Lisp together with the simplicity of the lambda-calculus have proven extremely powerful, something easily overlooked by programmers of languages with a fixed syntactic surface.

While other languages hide the evaluation (compilation) of code behind a fixed syntax-parser and cannot be extended from within, Lisp is modelled in its own data-structures and exploits the von-Neumann-concept of "code = data" (memory) fully. In fact an executable Lisp defined in itself can famously be described in half a page of a book.[19] This alone does not make Lisp functions pure, but Lisp was modelled for the lambda-calculus from the beginning and has avoided the mistake of introducing syntax, only to survive up until today as a weird outsider to all syntactical languages. [27] [21] These age through the baked-in semantic abstractions into the syntax only to regularly become a burden or plain history and force the need for new syntactical descriptions in syntactical successor languages to reflect new semantic abstractions and concepts. Even if these can be added to the syntax, the programmer has to wait for the language-designer and is bound to all decisions made by him or her.

A more detailed introduction to functional programming in the lambda-calculus, developping all basic functions and abstractions in 22 pages (without Lisp syntax) and resulting in a lazily evaluated AI for general board games, is the paper "Why Functional Programming Matters"[15]. We recommend unfamiliar readers to read it as a background to the following work. For now we pick two elementary lambda functions replacing imperative loops with recursive declarations.

While they are defined in terms of declarative mathematical induction in the paper, here they are already ported to Clojure's syntax to help understand the following code. Sequences can be dealt with as a pair of head and tail which is the traditional recursive building-block of a list in Lisp (often called cons-cell because of the synthetic function cons appending elements as heads to a list). We call semantic constructs of Lisp forms. The first symbol in each form delimited by "(" and ")" names the function the rest its parameters. If the parameters are forms themselves, they are evaluated **before** the outer form is evaluated. We can see how a recursive form unfolds into itself. The outer *fn*-forms are special (translated by compile-time macros) and allow to define a *lambda*, the parameter list is in "[" "]" marking a Clojure vector literal (data, not a function-application). Clojure further supports map literals in "{" "}" as seen later.

```
; f is a function, res is the intermediate result so far
; and seq is a sequence
(fn reduce [f res seq]
   (if (empty? seq) ; conditional special form if
        res ; return result if seq is empty
        (reduce f (f res (first seq)) (rest seq))))) ; else branch

; map falls out of reduce already, which only combines
; sequences and recursion through functional composition
(fn map [f seq]
   (reduce (fn [res elem] (cons (f elem) res))
           '()
           seq))
```

Note that both pure functions, *reduce* and *map*, the outer *fn*-forms, are defined anonymously, their name is only defined internally to allow recursion. We will use the *def* and *defn* (a shortcut for functions) to bind symbols to variables later. This happens dynamically in Lisp and hence introduces global mutable state by redefining symbols during runtime (called *vars* in Clojure). This is intentionally done to allow live-programming and a tight feed-back loop to the programmer, but has to be understood as problematic when used to manage application state and can easily break the pureness of the lambda-calculus and hence its nice properties. [20]

1.5.1 Persistent data-structures

The theoretical development of computer science together with studies of functional concepts since the 70s has in the last 10 years lead to practical implementations of persistent data structures. Persistent, or *immutable*, data structures, not to be confused with persistency on disk, but only in memory, allow to share structure between subsequent alterations of the same data structure while keeping the old value intact and hence only consume some more memory by attaining the access complexity properties, e.g. persistent hash-maps allow constant time lookup and assignment. They are implemented as different forms of trees. While this seems unrelated to functional programming, it marks a critical progress as it allows to share these data-structures in different pure functions conceptually as values (instead of memory addresses) while being able to efficiently change them in

parallel in some other part of the program, hence decoupling readers from writers and allowing pure functions per default without defensive copying every time a function is applied. This directly addresses criticism two.

Clojure is the first language to implement persistent data-structures as its native data-structures and brings much more achievements of research to practical application. But most significantly it allows *value-oriented* programming, where the programming can be modelled as a flow of values from an input to an output and allowing to scale-out with multithreading through lock-free reading of software-transactional-memory and saving values consistently (not writing memory directly and not locking either). In fact this was and is its center design motive.[12]

2 Setup

2.1 Introducing a modern functional Lisp with Clojure

Clojure is written for the JVM (and runs on CLR, Javascript, Python, Lua, Scheme and C). It has been designed as a hosted language, meaning it does not have primitive types of its own, but rather uses the ones of the runtime host. That way no additional abstractions forcefully add performance overhead as is the case for Jython or JRuby for instance. Clojure has number-boxing (with a numerical tower) and uses reflection, but both can be manually side-stepped in performance critical paths, directly addressing criticism 2. This also makes access to "native" constructs and libraries seemless, especially since macros can be used to statically compile optimized native paths when needed. This is especially important in game development, where the "inner-loop" found by profiling benefits heavily from low-level optimizations. [10]

2.2 Lazy evaluation

Clojure also covers lazy evaluation which allows to compute only parts needed at runtime depending on input-parameters to functions while modelling infinite sequences as a general solution. Lazy evaluation has been pioneered by Haskell (see also [15] for an introduction) to the point where all evaluations are done lazily, including function-parameter evaluation. This is uncombinable with side-effects, as sometimes parameters might be evaluated (executed), while sometimes they might not or in a different order.

Since the intent is to reduce sequential necessities between evaluations, it cannot be enforced in a lazily evaluated context. Clojure therefore doesn't enforce pure programming and uses strict evaluation (always evaluating the parameters before applying them to the function), giving the programmer still all the power he wants to insert side-effects whenever he or she needs them, even if only during life-development for exploration.

2.3 Potential of Clojure beyond the basics

Clojure has drawn designs from Common-Lisp, Scheme, Standard-ML, Haskell, Erlang, Ruby and other languages. Lisp is in that sense not dogmatic at all, on the contrary its minimalistic design is adaptable to almost anything the programmer (not the language-designer!) can imagine.[21]

Clojure also has sparked implementations of "Prolog" [6] usable for game AI and "Go"-like *communicating sequential processes* [5][4] (light-parallelism for distributed systems) just to name two general programming paradigms that can come in handy for our distributed game programming.

Its object-orientation features include *dynamic* dispatch steered by the programmer via *multimethods*, ad-hoc *polymorphism* even for uncontrollable alien/native types only matched by Common-Lisps CLOS implementation of the Metaobject-Protocoll[17]. Seperately type *inheritance* hierarchies can be defined ad-hoc by the programmer differently in different contexts. Clojure supersedes Common-Lisp in object-orientation though because Clojure's implementation "primitives" are protocol abstractions themselves, even the functions and sequences (data-structures like the omni-present building-block, the list) are used as protocols and hence can be polymorphically reinterprated. For example an outside "native" Java-class of our game library could be extended to the *IFn* function protocol to be usable whereever a lambda is expected. This polymorphism is more powerful than forced OO-paradigms and for this very reason interestingly mostly invisible.

The point is that we **don't** use any of these latter features or programming concepts in our prototype, but it is noteworthy that functional programming in Lisp allows to embed any programming paradigm in any S-expression "()" totally controlled by the programmer and can be selected separately a-la-carte instead of enforcing e.g. object-oriented syntax and semantics on all developpers of the system as in Java, C++ or C#. A whole range of bugs by incidental complexity between necessary language con-

structs and messy "boilerplate"-code can be avoided without loss of power. This also separates Lisp from all syntactical functional languages.[13]

2.4 Environment

Since an already implemented augmented-reality game, Ingress, uses *libgdx* and libgdx is a quite popular cross-platform open-source gaming-library covering the Android mobile audience we target, we chose it for gaming side-effects. We will use functional zippers (clojure.zip) to extract building data from OpenStreetMap-data in XML form. We also make use of Clojure's numeric tower. Other than that we use what the JVM and Clojure provide.

3 State-Management

3.1 Separating State from side-effects

We have come to our setup to allow functional programming and keep the logic of our process pure and decoupled from any side-effects. To make state changes explicit we make it *first-class* and model the whole environment of the game as a *map*-value we pass around. Some functions will perform transactions on this state, while all other functions are just called with portions of it and don't affect it (write to any memory as this map is the only shared memory conceptually). This very simple concept allows us already to have a stateless runtime (at least we can reconstruct its state just by knowing the value of the state-map) and have explicit points of synchronisation. Since changes to the state-map are executed through software-transactional-memory, we can separate gaming input (mostly user-movement for now) and simulation of the game world and run them in parallel. Rendering does not perform side-effects but just renders state appropriately, so we can keep it a completely separate concern.

```
(def state-demo1
  {:emanations [{:position {:x 0.8 :y 0.3}
                 :speed {:x 0 :y 0}
                 :force {:x 0 :y 0}
                 :inertia 0.5
```

```
                :sprite "emanation.png"}
               {:position {:x 0.1 :y 0.9}
                :speed {:x 0 :y 0}
                :force {:x 0 :y 0}
                :inertia 1
                :sprite "emanation.png"}]
   :players {:tom {:position {:x 0.5 :y 0.5}
                   :speed {:x 1 :y 1}
                   :sprite "player.png"
                   :inertia 0.8}}
   :symbols [{:position {:x 0.1 :y 0.8}
              :inertia 1
              :sprite "water.png"}]
   :buildings (load-buildings)
   :hud (fn [{:keys [emanations players symbols buildings resolution]}]
          (str "Press s for new symbol\n"
               "Press e for new emanation\n"
               (count emanations) " emanations \n"
               (count players) " players \n"
               (count symbols) " symbols \n"
               (count buildings) " buildings \n"
               "resolution: " resolution))

   :resolution {:x 800 :y 480}})
```

This demo is mutably bound to the var "state-demo1" for live-coding purposes, but is not accessed through this var but passed to the different systems in the software-transactional-memory primitive "atom", which coordinates writes locklessly through the "swap!"-function:

```
(def test-state (atom state-demo1))
; start rendering and input handling
(start test-state)
; control (pure) simulation
(sim/start test-state)
(sim/stop test-state)
; control server
(serv/start test-state)
(serv/stop test-state)
```

```
; control client connection
(cli/start test-state)
; Change game-state safely, this is also used to refresh state from the
; server
(swap-state! test-state (fn [old] state-demo1))
```

3.2 Game simulation

The game simulation has to take the value of the state-map and calculate the new position of players and "monsters", called emanations in projectR:

```
(defn simulate!
  "Simulation loop in dependence of delta time."
  [state last-millis]
  (let [current-millis (System/currentTimeMillis)
        delta-time (/ (- current-millis last-millis)
                      1000.)]
    (swap! state (fn [old]
                   (-> old
                       (emanate delta-time)
                       (decay-symbols delta-time))))
    (Thread/sleep 50)
    (if (::run (:volatile @state))
      (recur state current-millis))))
```

It is run in a separate thread and can be controlled through the "::run" value in a "volatile" area of state which is used for runtime state. It can also be stopped and started that way independently (implementing the headless server). All values under ":volatile" are omitted on transfers or state saving because they are either present or can be reconstructed on-the-fly on runtime (= cache). It is logically also the only part of the map not being serializable.

All simulation functions being called by "simulate!" just return the new value and are pure then:

```
(defn emanate [state delta-time]
  (let [{:keys [players symbols emanations buildings]} state
        emanies (map #(update-in % [:inertia] * -0.1) emanations)]
```

```
    (assoc state :emanations (pmap (partial move-emanation
                                            buildings
                                            (concat symbols
                                                    (vals players)
                                                    emanies)
                                            delta-time)
                                   emanations))))
```

The "pmap" function allows to transparently run the "map"-function in parallel without additional care (because we have omitted shared state between each call), so our simulation can make use of multicore and scale linearly. The actual gravitational physics between players and emanations is done in "move-emanation" and "attr-force":

```
(defn move-emanation [buildings attractors dt emanation]
  (let [{:keys [speed position inertia]} emanation
        decay (math/expt 0.8 dt)
        total-force (reduce (partial merge-with +)
                            (map (partial attr-force emanation)
                                 attractors))
        etf (eucl-dist total-force)
        thresh 0.00002
        nsp (if (> etf thresh)
              (merge-with +
                          (scale decay speed)
                          (scale (/ decay (* inertia 10)) total-force))
              (scale decay speed))]
    (assoc emanation
      :force total-force
      :speed nsp
      :position (if-not (colliding? buildings position nsp)
                  (merge-with + position nsp)
                  position))))


(defn attr-force [source target]
  (let [spos (source :position)
        tpos (target :position)
        sert (source :inertia)
```

```
        tert (target :inertia)
        attr-const 0.00005
        d (dist spos tpos)
        ed (eucl-dist d)
        nd (scale (/ 1 (max ed 0.0001)) d)
        f (/ (* sert tert attr-const)
             (max 0.8 (* ed ed)))]
    {:x (* f (nd :x))
     :y (* f (nd :y))}))
```

There is also collision detection for buildings implemented, but we spare the details as this is just to illustrate how concerns are separated.

3.3 Inputs

Inputs are taken from the libgdx-API by implementing its OO-interface for *ApplicationListener*:

```
(defn- libgdx-listener [state]
  (reify ApplicationListener
    (create [this]
      (init-render! state))
    (pause [this])
    (resume [this])
    (render [this]
      ; this guarantees continuous movement
      (input-key-move! state)
      (input-touch-move! state)

      (clear-plane!)
      (update-camera! state)
      (draw-all! state))
    (resize [this width height]
      (println "size change: " width " " height)
      (swap! state (fn [old] (assoc old :resolution
                                    {:x width :y height}))))
    (dispose [this]
      (dispose-render! state)
      (swap! state #(dissoc % :volatile))
```

```
      (System/exit 0))))

(defn- input-touch-move! [state]
  (if (.isTouched Gdx/input)
    (let [{:keys [volatile resolution]} @state
          width (resolution :x)
          height (resolution :y)
          touch-pos (Vector3.)
          x (.getX Gdx/input)
          y (.getY Gdx/input)]
      (.set touch-pos x y 0)
      (swap! state (fn [old] (assoc-in old
                                       [:players :tom :position]
                                       {:x (/ x width)
                                        :y (- 1 (/ y height))}))))))))
```

3.4 Rendering

Rendering is also called in the "render!" function, but it does not affect the state-map, besides volatile caches called "memoization" in functional programming because pure functions getting the same parameters always return the same value:

```
(defn draw-all!
 "Glue it all together."
 [state]
 (let [{:keys [players emanations symbols volatile debug-triags]} @state
       {poly-batch ::poly-batch sprite-batch ::sprite-batch} volatile]

   (doall (map (fn [{:keys [position sprite inertia]}]
                 (draw-sprite! state sprite-batch sprite position inertia))
               (concat (vals players) symbols emanations)))

   (draw-triags! state poly-batch "building.png" (btriags-cache state))

   (draw-hud! state sprite-batch)))
```

Actual drawing is implemented through the platform-independent libgdx-API:

```
(defn- draw-sprite! [state ^SpriteBatch batch sprite position transparency]
  (.begin batch)
  (let [{:keys [x y]} (as-game-coords (:resolution @state) position)
        sp (texture-cache state sprite)
        w (.getWidth sp)
        h (.getHeight sp)]
    (.setColor batch (float 1) (float 1) (float 1) (float transparency))
    (.draw batch sp (float (- x (/ w 2))) (float (- y (/ h 2)))))
  (.end batch))
```

We omit the constant drawing of buildings and other rendering functions as they basically look the same. We can also see here how nicely the host can be called without additional abstractions. *SpriteBatch* for instance is a libgdx type and is type-hinted here to avoid reflection as measured by profiling and Clojure detection of reflection. It is not handled differently than Clojure objects, except it has mutable state cared about by libgdx.

4 Conclusion

We have not dived into the technical details as these are not the point of this functional prototype. While we use very sophisticated functional abstractions and helpers like persistent data-structures and software-transactional-memory in different places, performance has proven bound by the square complexity of game-physics (gravitation) only. We have transparently implemented memoization in rendering to avoid recomputation in every frame-rendering step and bring CPU usage down, but it is remarkable that even before these optimization the prototype was playable at 60 FPS. This project has proven to the author that Clojure has no performance penalty if it is optimized carefully with a profiler. This is long known in the Clojure community and defeats criticism 2. [10] We haven't exploited static compilation with macros yet in the prototype though.

The decoupling of state and functions has already allowed to implement a server- and client-side in the same code and swap in server updates or different scenarios from the outside during runtime without "synchronisation" or ORM. The only necessary step was to make the state explicit and serializable, so it can be loaded over any communication protocol and swapped into clients repeatedly, while allowing them to simulate based on their partial knowledge of state to give a fluent impression to the player.

State can be decoupled further with messaging as done by "Pedestal".[23] This framework also uses deltas to steer the renderer (read: cause minimal side-effects), something that can easily be done later on the state-map in the prototype as well with the provided function "diff" to optimize the entropy sent over the wire to mobile clients.

Chosing a proper state-model [20] further formalized by "Pedestal" also addresses the fact that Lisp is so flexible it can be its own obstacle, [26] like a piece of mud,[3] by formalizing state in a standard way and allowing to implement only pure functions, which are standardized and a simple building block. The glue functions altering state like "simulate!" and "draw-all!" we still had to write fall away then. While this prototype does not address the problems of distributed state in total yet, it models a logically simple, consistent and scalable implementation of the game scenario and distributed applications in general. It is also noteworthy that through the separation of side-effects it becomes possible to easily replace the gaming-library or any other part by just reimplementing the according isolated side-effecting places in the code like the "render"-namespace. This is the highest form of implicit portability. All good things fall out of chosing a *logically* simple computational primitive, the *lambda*.[13, 8]

5 Literature and Resources

References

[1] *Alan Cox*. Accessed: 2013-08-22. URL: https://en.wikiquote.org/wiki/Alan_Cox.

[2] *Clojure at Nokia Entertainment*. Accessed: 2013-08-22. URL: http://skillsmatter.com/podcast/scala/clojure-at-nokia-entertainment/te-6533.

[3] *Clojure in the Large*. Accessed: 2013-08-25. URL: http://www.infoq.com/presentations/Clojure-Large-scale-patterns-techniques.

[4] *ClojureScript core.async dots game*. Accessed: 2013-08-20. URL: http://rigsomelight.com/2013/08/12/clojurescript-core-async-dots-game.html.

[5] *core.async*. Accessed: 2013-08-20. URL: https://github.com/clojure/core.async.

[6] *core.logic*. Accessed: 2013-08-20. URL: https://github.com/clojure/core.logic.

[7] *Curry–Howard correspondence*. Accessed: 2013-08-22. URL: https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence.

[8] Julie Ecklar. *Eternal Flame*. Accessed: 2013-08-25. URL: http://www.gnu.org/fun/jokes/eternal-flame.html.

[9] *Forth*. Accessed: 2013-08-22. URL: https://en.wikipedia.org/wiki/FORTH.

[10] *Functional Fluid Dynamics in Clojure*. Accessed: 2013-08-22. URL: http://www.bestinclass.dk/index.clj/2010/03/functional-fluid-dynamics-in-clojure.html.

[11] Brian Goetz et al. *Java Concurrency in Practice*. en. Addison Wesley Professional, 2006.

[12] Rich Hickey. *Keynote: The Value of Values*. Accessed: 2013-08-25. URL: http://www.infoq.com/presentations/Value-Values.

[13] Rich Hickey. *Simple Made Easy*. Accessed: 2013-08-25. URL: http://www.infoq.com/presentations/Simple-Made-Easy.

[14] Greg Hoglund and Gary McGraw. *Exploiting Software - How to Break Code*. en. Addison Wesley, 2004.

[15] John Hughes. "Why Functional Programming Matters". en. In: (1990). URL: http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

[16] *Java-Performance*. Accessed: 2013-08-22. URL: https://en.wikipedia.org/wiki/Java_performance.

[17] Georg Kiczales, Jim des Revières, and Daniel G. Boborow. *The Art of the Metaobject Protocol*. en. Camebridge, Massachusetts and London England: The MIT Press, 1991. ISBN: ISBN 0-262-11158-6.

[18] *Lisp machine*. Accessed: 2013-08-22. URL: https://en.wikipedia.org/wiki/Lisp_Machine.

[19] John et al. McCarthy. *LISP 1.5 Programmer's Manual*. en. The MIT Press, 1985, p. 13. URL: http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf.

[20] *My Clojure Workflow, Reloaded*. Accessed: 2013-08-25. URL: http://thinkrelevance.com/blog/2013/06/04/clojure-workflow-reloaded.

[21] David Nolen. *An Introduction to Lisp Macros*. Accessed: 2013-08-20. URL: http://vimeo.com/23907627.

[22] *Overtone*. Accessed: 2013-08-22. URL: http://overtone.github.io/.

[23] *pedestal*. Accessed: 2013-08-25. URL: http://pedestal.io/.

[24] *Quil*. Accessed: 2013-08-22. URL: https://github.com/quil/quil.

[25] *Technology Radar*. Accessed: 2013-08-22. URL: http://www.thoughtworks.com/radar.

[26] *The Bipolar Lisp Programmer*. Accessed: 2013-08-22. URL: http://www.lambdassociates.org/blog/bipolar.htm.

[27] *The Hundred-Year Language*. Accessed: 2013-08-22. URL: http://www.paulgraham.com/hundred.html.

[28] *Video Lecture oriented on "Structure and Interpretation of Computer Programs"*. Accessed: 2013-08-22. URL: http://www.youtube.com/watch?v=2Op3QLzMgSY.