A History of Clojure

- https://download.clojure.org/papers/clojure-hopl-iv-final.pdf

Introduction

Clojure의 목적은 다음과 같이 가장 간결하게 요약될 수 있습니다. 저는 Java 또는 C#만큼 수용 가능한 언어를 원했지만, 제가 전문적으로 해왔던 정보 시스템 개발 종류에 사용하기 위해 훨씬 더 간단한 프로그래밍 모델을 지원하기를 원했습니다. 저는 은퇴 저축으로 자금을 조달한 안식년인 2005년에 Clojure 작업을 시작했습니다. 안식년의 목적은 결과, 상업적 가능성 또는 다른 사람들의 의견에 상관없이 내가 흥미롭게 생각하는 모든 작업을 수행할 수 있는 기회를 제공하는 것이었습니다. 이것이 Lisp 또는 함수형 언어 작업을 위한 전제 조건이라고 말할 수 있습니다. 나는 2년 동안의 자기 주도적 작업을 위해 예산을 책정했고 Clojure는 내가 추구했던 두 가지 프로젝트 중 하나였습니다. 약 1년 후에 다른 프로젝트(와우 각 모델링 및 기계 청취 문제)는 2~5년이 더 필요할 수 있는 연구 노력에 더 가깝다고 결정했기 때문에 그 시점에서 Clojure를 유용한 상태로 만드는 데 전념했습니다. 저는 2007년 가을에 오픈소스 프로젝트로 Clojure의 첫 번째 버전을 발표하고 출시했습니다. 이 기간 동안 100% 구현 작업을 수행했으며 2010년 1.2 릴리스를 통해 90% 이상 구현했습니다. Clojure 출시 이후 커뮤니티의 피드백, 제안 및 노력으로부터 큰 도움을 받았습니다. 저는 커뮤니티에서 “ 인생을 위한 자비로운 독재자(BDFL)로 인정받고 있으며 진화와 관련된 모든 결정을 계속 내립니다. Clojure는 다른 사람들의 훌륭한 아이디어로 가득 차 있지만, 그 결점에 대해서는 나 혼자만 책임을 진다. [1)]

BACKGROUND AND MOTIVATION

Tuesday Afternoon

저는 1987년부터 상용 소프트웨어 개발을 하고 있었습니다. 거의 항상 개발 책임자이자 기본 설계자였습니다. 처음에는 C++로, 그 다음에는 Java와 C#으로, 당시에는 일반적이었고, 제도적 관성에 따라 30년이 지난 지금도 여전히 그렇습니다. 저는 스케줄링 시스템, 방송 자동화, 수익 관리, 오디오 인식, 출구 투표 표 작성 및 선거 예측 등을 작업했습니다. 나는 내 작업과 전문 프로그래머가 가장 일반적으로 수행하는 작업을 정보 시스템 프로그래밍으로 정의할 것입니다. 대부분의 개발자는 주로 정보(세계에 대한 사실)를 획득, 추출, 변환, 유지 관리, 분석, 전송 및 렌더링하는 시스템을 만드는 데 종사하고 있습니다. 대부분의 경우 이 정보는 고객, 공급업체, 광고주, 여행자, 유권자, 회원, 학생, 환자 등의 인간 활동을 문서화하고 모든 부정 행위를 처리해야 합니다. 이것은 완전히 열거된 공간에서 자체 규칙을 구성하는 프로그래밍 언어 컴파일러와 같은 인공 시스템과 완전히 대조적이며, 불규칙성을 제거하고 일치하지 않는 모든 것을 거부할 수 있습니다. 정보 시스템 프로그래머는 준수를 거부하는 정보와 실제 활동에 약간의 규칙적인 모델을 중첩하려고 시도하는 감사할 수 없는 작업이 있습니다. 예를 들어, 음악 스케줄링에서 아티스트가 노래를 가지고 있는지 아니면 노래에 아티스트가 있는지 결정하려고 합니다. 또는 각 아티스트의 노래가 인접한 쌍으로 재생되어야 하는 '두 번의 화요일'을 제외하고 각 아티스트의 노래 재생을 분산하도록 스케줄러를 최적화합니다. 정보 시스템 프로그래밍에서 두 개의 화요일은 어디에나 있습니다. [2)]

I'm a Believer

1990년대 중반까지 저는 C++ 전문가였으며 NYU에서 보조 C++를 가르쳤고 정적 타이핑의 이점을 지지하고 옹호했습니다(하지만 절충안을 무시했습니다, 죄송합니다!). 나는 행복하게 유형 매개변수화 트릭[Hickey 1996]을 발견하고, const-correct shop 등을 운영했습니다. 그러나 시간이 지남에 따라 정보 시스템 프로그래밍에 대한 이러한 정적으로 유형이 지정된 클래스 모델의 작업 적합성은 매우 낮았고 유

형 검사의 이점은 특히 프로그래머가 직면한 가장 큰 실제 문제를 해결할 때 최소화되었습니다. 명령형, 상태 저장 프로그래밍에 내재된 압도적인 복잡성. 프로그램이 커짐에 따라 상태와 관계에 대한 모든 가정을 유지하면서 동시에 변경하려는 엄청난 노력이 필요했습니다. 그리고 우리는 특수성(비록 모범 사례 캡슐화, 추상화 및 매개변수화에도 불구하고)으로 인해 잠식하고 결국에는 무력화, 결합 및 거대한 코드베이스에 직면했습니다. 한 시간 이상의 C++ 빌드가 일반적이었습니다. Clojure 작업 직전 몇 년 동안 저는 미국의 전국 출구 조사에 사용할 시스템에 대해 작업했습니다. 이 시스템에는 대규모 통계 모델을 자동화하는 작업이 포함되었습니다. 초기에 우리는 통계를 수행하는 필수 접근 방식이 수학적이지 않고 시스템 전체에서 예상되는 동시성으로 인해 부적합하다고 결정했습니다. 작업에 대한 F#을 탐색한 후(그리고 표현이 불충분하다는 것을 발견한 후) 우리는 시스템의 나머지 부분과 마찬가지로 C#으로 통계를 코딩하기로 결정했습니다(이곳은 Microsoft .Net 상점이었습니다). 통계 사양의 수치와 함께 사용할 수 있는 사용자 지정 변경할 수 없는 데이터 구조(C#을 모르는 통계학자가 코드를 보고 이해할 수 있도록)와 구조를 조작하고 수행하기 위한 기능 라이브러리를 설계했습니다. 계산. 이것은 우리가 걱정하지 않았으며 여전히 걱정하지 않는 훨씬 간단한 코드를 생성하여 큰 성공을 거두었습니다. 그러나 결과 C# 코드는 신규 및 숙련된 C# 개발자의 눈에 기괴하고 비관용적이었습니다. 따라서 함수형 프로그래밍은 승리했지만 비 FP 언어의 FP는 널리 적용되는 것을 볼 수 없었습니다. [3)]

Who Do You Love?

나는 여가 시간에 Lisp, Prolog 및 Smalltalk를 가지고 노는 언어 괴짜였습니다. 2000년에 독립 컨설턴트가 되었을 때 나는 Common Lisp[Steele Jr et al. 1990] 그리고 그 안에 몇 가지 실제 시스템을 작성했습니다. 그것은 계시였습니다. 불필요한 복잡성의 거대한 층이 단순히 사라졌습니다. 문제에 필요한 만큼의 언어를 사용할 수 있는 유연성이 있었습니다. 도메인과 직접 관련된 코드의 비율이 증가했습니다. 개발은 훨씬 더 빨랐고 결과 프로그램은 더 일반적이고 변경하기 쉬웠습니다. 내 경력 내내 C++/Java/C#을 사용하면서 내가 " 잘못하고 있다 " 는 느낌을 피할 수 없었습니다. Common Lisp와 같은 다른 선택이 필요했습니다. 그렇지 않으면 전문 소프트웨어 개발자로 계속할 수 없었습니다. 언어 기능이 중요하지만 전문가의 언어 채택에 대한 주요 장애물은 개발자와 이해 관계자의 수용 가능성입니다. 따라서 Clojure는 내가 함께 일할 개발자와 내가 일할 회사, 그렇지 않으면 생계를 유지하는 데 사용할 수 없는 회사에 최소한 실용적이어야 했습니다. 나는 이것을 디자인의 의제 항목으로 삼았지만 사전에 그들의 의견을 구하려는 동기가 아니었습니다. 당시 Lisp에 대한 전문 개발자의 조언은 " 죽었다 " 이고 함수형 프로그래밍에 대한 조언은 " 저게 뭐야? " 였기 때문입니다. 새로운 프로그래밍 언어를 작성하는 것은 " 당신 미쳤어요 " 였습니다.[4)]

2005년은 ISV가 주로 C/C++를 사용하고 많은 기업이 스스로를 Java 또는 .Net '상점'으로 간주하는 등 상용 소프트웨어 개발에서 언어 다양성의 최하점을 나타냈을 수 있습니다. 이것은 웹 개발을 위한 동적 언어(예: Ruby on Rails)가 등장하기 전이었고 많은 회사에서 JVM이나 CLR에서 실행되지 않는 모든 것을 배포하고 운영하는 것을 꺼렸습니다. 나는 Common Lisp에서 두 번 상업 작업을 했습니다. 스케줄링 시스템과 수율 관리 시스템입니다. 처음으로 프로그램을 배포할 수 있도록 C++로 다시 작성해야 했을 때, 초기 개발보다 시간이 오래 걸리고 훨씬 더 많은 코드가 필요했으며 실행 속도가 훨씬 빠르지 않은 힘든 작업(나는 노련한 C++ 프로그래머였지만) . 두 번째 경우에도 Common Lisp는 배포에 적합하지 않았으므로 클라이언트에 전달하고 런타임 실행을 위해 SQL 저장 프로시저를 생성하도록 프로그램을 설계했습니다. 2005년에 Clojure를 시작하기 전에 저는 JVM과 Common Lisp의 실용성과 수용 가능성을 높이기 위해 JVM과 Common Lisp 사이에 다리를 만들려고 여러 번 시도했습니다. 이들은 DotLisp[Hickey 2003], CLR에 대한 호스트 상호 운용성을 포함하는 해석된 Lisp, CL에 JVM을 임베드한 라이브러리인 jFli[Hickey 2004], 유사한 API를 노출했지만 Foil[Hickey and Thorsen 2005]이었습니다. CL 런타임과 JVM 간에 IPC를 사용했습니다. 이들 중 어느 것도 프로덕션 수준의 솔루션을 제공하지 않았지만 Clojure의 궁극적인 호스트 구문을 확실히 알리고 Clojure가 호스팅된다는 아이디어를 제공했습니다. [5)]

관련 문서

- Clojure

[1)]
The objective for Clojure can be summarized most succinctly as: I wanted a language as acceptable as Java or C#, but supporting a much simpler programming model, to use for the kinds of information system development I had been doing professionally. I started working on Clojure in 2005, during a sabbatical I funded out of retirement savings. The purpose of the sabbatical was to give myself the opportunity to work on whatever I found interesting, without regard to outcome, commercial viability or the opinions of others. One might say these are prerequisites for working on Lisps or functional languages. I budgeted for two years of self-directed work, and Clojure was one of two projects I pursued. After about a year I decided the other project (a cochlear modeling and machine listening problem) was more of a research endeavor that might require two to five more years, so I dedicated myself at that point to getting Clojure to a useful state. I announced and released the first version of Clojure, as an open source project, in the fall of 2007. I did 100% of the implementation work during this time, and more than 90% through the 1.2 release in 2010. Subsequent to release Clojure benefited greatly from the feedback, suggestions and effort of its community. I am accepted by the community as “benevolent dictator for life” (BDFL) and continue to make all decisions relating to its evolution. Clojure is full of the great ideas of others, but I alone take responsibility for its faults.
[2)]
I had been doing commercial software development since 1987, almost always as the development lead and primary architect, first in C++, then Java and C#, as was common then and, in testament to institutional inertia, is still now thirty years later. I worked on scheduling systems, broadcast automation, yield management, audio recognition, exit poll tabulation and election projection et al. I would broadly characterize my work, and the work most commonly done by professional programmers, as information systems programming. Most developers are primarily engaged in making systems that acquire, extract, transform, maintain, analyze, transmit and render information—facts about the world. Most often, this information documents some human activity, be that of customers, suppliers, advertisers, travelers, voters, members, students, patients etc and must deal with all the irregularity thereof. This is in stark contrast to artificial systems, e.g., programming language compilers, which make up their own rules, in fully enumerated spaces, can eliminate irregularity and can reject anything which does not conform. Information system programmers have the thankless task of attempting to superimpose some- what regular models over information and real-world activity that refuses to comply. For instance, in music scheduling, trying to decide: whether artists have songs or songs have artists. Or optimizing a scheduler to spread out the plays of songs by each artist, except on ‘twofer Tuesdays’ when songs by each artist must be played in adjacent pairs. In information systems programming, twofer Tuesdays are everywhere.
[3)]
By the mid 1990’s I was a C++ expert, taught advanced C++ as an adjunct at NYU, and was a proponent and advocate of the benefits of static typing (but neglected the tradeoffs, sorry students!). I was happily discovering type parameterization tricks [Hickey 1996], running a const-correct shop etc. However, over time, in my experience, the suitability-to-task of these statically typed class models for information systems programming was quite low, and the benefits of the type checking minimal, especially in addressing the number one actual problem faced by programmers: the overwhelming complexity inherent in imperative, stateful programming. As programs grew large, they required increasingly Herculean efforts to change while maintaining all of the presumptions around state and relationships, never mind dealing with race conditions as concurrency was increasingly in play. And we faced encroaching, and eventually crippling, coupling, and huge codebases, due directly to specificity (best-practice encapsulation, abstraction, and parameterization notwithstanding). C++ builds of over an hour were common. In the years immediately preceding work on Clojure I worked on

the system to be used for the national exit poll in the U.S. This system involved automating large statistical models. Early on we decided that an imperative approach to doing the stats was a misfit, being not mathematical and also due to the expected concurrency throughout the system. After exploring F# for the task (and finding it insufficiently expressive), we decided to code the stats in C# like the rest of the system (this was a Microsoft .Net shop). I designed some custom immutable data structures whose use would be co-aligned with the figures in the statistical specifications (so statisticians who did not know C# could look at the code and understand it), and a functional library for manipulating the structures and doing the calculations. This was a great success, yielding much simpler code that we did not, and still do not, worry about. However, the resulting C# code was, in the eyes of both new and experienced C# developers, bizarre and non-idiomatic. Thus functional programming was a win, but FP in a non-FP language was not something I could see being widely applied.
4)

I had always been a language geek, playing with Lisp, Prolog and Smalltalk in my spare time. When I became an independent consultant in 2000, I spent more time with Common Lisp [Steele Jr et al. 1990] and wrote a couple of real systems in it. It was a revelation. Huge layers of unnecessary complexity simply vanished. I had the flexibility to use exactly as much language as was needed for the problem. The percentage of code directly related to the domain increased. Development was much faster, the resulting program was more general and easier to change. It was impossible to avoid the sinking feeling that I had been “doing it wrong” by using C++/Java/C# my whole career. I needed another choice more like Common Lisp or I wouldn’t be able to continue as a professional software developer. While language features matter, the primary hurdle to language adoption by professionals is acceptability to developers and stakeholders. Thus, Clojure did have to be at minimum practical for developers I would work with, and companies I might work for, or else I couldn’t use it to make a living. I took this as an agenda item for its design but not as motivation to seek their input in advance, because at the time the advice from professional developers about Lisp was “it’s dead” and about functional programming was “what’s that?”, and about writing a new programming language was “you’re crazy”.
5)

The year 2005 may have represented the nadir of language diversity in commercial software development, with ISVs primarily using C/C++ and many businesses considering themselves either Java or .Net ‘shops’. This was before the ascent of dynamic languages for web development (e.g., Ruby on Rails), and many companies were reluctant to deploy and operate anything that didn’t run on the JVM or CLR. I did commercial work in Common Lisp twice - a scheduling system and a yield management system. The first time the program had to be rewritten in C++ in order to be acceptable for deployment, an arduous task (though I was a seasoned C++ programmer) that took longer than the initial development, was much more code, and did not run significantly faster. In the second case again Common Lisp was not acceptable for deployment, so I designed the program to generate SQL stored procedures for delivery to the client and runtime execution. Prior to embarking on Clojure in 2005, I had made several attempts to create a bridge between the JVM and Common Lisp, in order to increase the latter’s practicality and acceptability. These were DotLisp [Hickey 2003], an interpreted Lisp with host interop for the CLR, jFli [Hickey 2004], a library that embedded a JVM in CL, and Foil [Hickey and Thorsen 2005], a library that exposed a similar API but used IPC between the CL runtime and the JVM. None of these yielded production- level solutions, but they definitely informed Clojure’s ultimate host syntax and fed the idea of Clojure being hosted.