Report from Dagstuhl Seminar 12011

Foundations for Scripting Languages

Edited by

Robert Hirschfeld[1], Shriram Krishnamurthi[2], and Jan Vitek[3]

1 **Hasso-Plattner-Institut, Potsdam, DE,** hirschfeld@hpi.uni-potsdam.de
2 **Brown University, Providence, US,** sk@cs.brown.edu
3 **Purdue University, US,** jv@cs.purdue.edu

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 12011 on the "Foundations for Scripting Languages". The choice of "for" rather than "of" is intentional: it is our thesis that scripting languages are in need of foundations to support their extensive use but lack them, and we hope this event consolidated and advanced the state of the art in this direction.

Seminar 02.–06. January, 2012 – www.dagstuhl.de/12011
1998 ACM Subject Classification D.2 Software Engineering, D.2.4 Formal Methods, D.3 Programming Languages, D.3.1 Semantics, D.3.4 Compilers, I.7.2 Scripting Languages
Keywords and phrases scripting languages, programming languages semantics, type systems, verification techniques, security analyses, scalability, rapid software development
Digital Object Identifier 10.4230/DagRep.2.1.1

1 Executive Summary

Robert Hirschfeld
Shriram Krishnamurthi
Jan Vitek

Common characteristics of scripting languages include syntactic simplicity, a lack of onerous constraints for program construction and deployment, the ability to easily connect to and control systems processes, strong built-in interfaces to useful external objects, extensive library support, and lightweight (and embeddable) implementations. More broadly, these characteristics add up to strong support for effective software prototyping. Due to a combination of these characteristics, common scripting languages like Perl, Python, Ruby, JavaScript, Visual Basic, and Tcl have moved from the fringes to mainstream program development.

To academics, these languages do not appear that different from, say, Scheme or ML. Since languages like Scheme and ML have well-defined semantics and other formal attributes, the mainstream passion for scripting languages may appear to simply be the result of ignorance of better languages amongst mainstream developers. However, the properties that scripting language users claim to find most beneficial are often *not* found in their more academic counterparts, such as a strong orientation towards systems process management, easily extensible objects, specific but useful control operators, etc.

In short, the academic tendency towards reductionism appears to miss some important characteristics. In particular, properties that may appear incidental—and are ignored by

the formalization of academic languages—may actually be essential. As a result, the formal study of scripting languages is a worthwhile research activity in its own right.

Not only does the study of scripting offer academics fresh problems, their results have the potential for widespread benefit. As scripts grow into programs, the very characteristics that seem an advantage sometimes prove to be disadvantages. If any object can be extended by any other object, it is impossible to reason about its behavior. If any object can access any resources, it is impossible to bound security implications. If programmers can places values of any type into a variable, it is impossible to obtain type guarantees. And so on. In other words, the very flexibility that enables prototyping inhibits the reasoning necessary for programs to grow in scale.

In the early days of scripting, there was an expectation that scripts were not meant to "grow up". Rather, as a prototype proved valuable, it would be turned into a program in a mainstream language, such as Java. However, reality does not match this vision. First, once a system becomes valuable to an organization, it is not possible to halt development on it while waiting for a full re-implementation. Second, even if the current version is converted to Java, the next version would probably still benefit from the benefits of prototyping. Thus, in both cases, programs that start in a scripting language are likely to remain in it. Finally, even if clients do want to rewrite the program in a more mature language, they would benefit from formal support to enable this conversion.

As a result, the formal study of scripting languages is a worthwhile research activity in its own right. In particular, we hope this seminar had both direct and indirect impact on academia and industry. We also hope that, based on our discussions, academics will identify concrete problems that need solutions and find scripting language experts who they can communicate with. In turn, we hope scripting experts identified knowledge, expertise, and interest from academia and are better aware of how to formulate problems for academics and map their solutions back to practice.

2 Table of Contents

3 Overview of Talks

3.1 Eval Begone!

Gregor Richards (Purdue University, US)

Eval is a common feature in dynamic languages, but an uncommon feature in analyses. Our work measures the real-world use of eval and determines its utility, in search of the "mythical" proper use of eval. We then introduce a system for the automated removal of eval by interactive analysis of its use and dynamic replacement with static suggestions.

3.2 Evaluating the Design of the R Language

Jan Vitek (Purdue University, US)

Joint work of Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek

R is a dynamic language for statistical computing that combines lazy functional features and object-oriented programming. This rather unlikely linguistic cocktail would probably never have been prepared by computer scientists, yet the language has become surprisingly popular. With millions of lines of R code available in repositories, we have an opportunity to evaluate the fundamental choices underlying the R language design. Using a combination of static and dynamic program analysis we can assess the impact and success of different language features.

3.3 Reasoning about Javascript

Philippa Gardner (Imperial College London, GB)

Joint work of Philippa Gardner, Sergio Maffeis, and Gareth Smith

JavaScript has become the most widely used language for client-side web programming. The dynamic nature of JavaScript makes understanding its code notoriously difficult, leading to buggy programs and a lack of adequate static-analysis tools. We believe that logical reasoning has much to offer JavaScript: a simple description of program behaviour, a clear understanding of module boundaries, and the ability to verify security contracts.

We introduce a program logic for reasoning about a broad subset of JavaScript, including challenging features such as prototype inheritance and with. We adapt ideas from separation logic to provide tractable reasoning about JavaScript code: reasoning about easy programs is easy; reasoning about hard programs is possible. We prove a strong soundness result. All libraries written in our subset and proved correct with respect to their specifications will be well-behaved, even when called by arbitrary JavaScript code.

3.4 Language Support for Third-party Code Extensibility

Benjamin Lerner (University of Washington, Seattle, US)

Browsers today support extensions, third-party pieces of script and markup that provide new or modified behavior for the underlying system. Likewise, users can inject scripts into web sites to modify them in a similar fashion. However, the idioms used to achieve this injection are cryptic, brittle, and have severe semantic flaws.

In this work we propose adding a new linguistic primitive to JavaScript, namely dynamic aspect weaving, that supports these extensions in a more robust, understandable, and stable way. As a side benefit, the new mechanism often out-performs the original idioms used.

3.5 Empirical Studies on Static vs. Dynamic Type Systems

Stefan Udo Hanenberg (Universität Duisburg-Essen, DE)

While static and dynamic type systems are exhaustively discussed by a large number a people, there is still no evidence whether (or in what situations) a static or dynamic type system provides a measurable benefit for software developers. This talk summarizes the results and the underlying ideas for an experiment series which compares the impact of static and dynamic type systems on software developers (based on the measurements of development time). The preliminary results so far are that the possible benefit of static and dynamic type systems is programming task specific. Furthermore, there is some evidence that type casts are no valid argument against static type systems.

3.6 Engineering a JavaScript Semantics

Arjun Guha (Brown University, Providence, US)

Joint work of Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

We reduce JavaScript to LambdaJS, a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

3.7 AmbientTalk as a Scripting Language

Theo D'Hondt (Vrije Universiteit Brussel, BE)

AmbientTalk is a language for mobile ad-hoc networks. It combines actors with effects and promotes failure to the rule rather than the exception. AmbientTalk and its implementation is described and subsequently compared to Python as a scripting language.

3.8 Life After main()

David Herman (Mozilla, Mountain View, US)

Scripting languages are often embedded in dynamic environments such as editors or browsers, and provide dynamic evaluation through REPL's. When a dynamic language contains static semantics such as static scoping, types, macros, or operator overloading, the interaction between the static and dynamic portions of the language can be fiendishly complicated. In my talk I discuss some of the surprising interactions and describe some of the design landscape for designing scripting languages with static semantics.

3.9 RubyX: Symbolic Execution for Security Analysis of Ruby on Rails

Jeffrey Foster (University of Maryland, College Park, US)

Joint work of Jeffrey Foster, Avik Chaudhuri, and Jong-hoon (David) An

Many of today's web applications are built on frameworks that include sophisticated defenses against malicious adversaries. However, mistakes in the way developers deploy those defenses could leave applications open to attack. To address this issue, we introduce Rubyx, a symbolic executor that we use to analyze Ruby-on-Rails web applications for security vulnerabilities. Rubyx specifications can easily be adapted to variety of properties, since they are built from general assertions, assumptions, and object invariants. We show how to write Ruby specifications to detect susceptibility to cross-site scripting and cross- site request forgery, insufficient authentication, leaks of secret information, insufficient access control, as well as application-specific security properties. We used Rubyx to check seven web applications from various sources against out specifications. We found many vulnerabilities, and each application was subject to at least one critical attack. Encouragingly, we also found that it was relatively easy to fix most vulnerabilities, and that Rubyx showed the absence of attacks after our fixes. Our results suggest that Rubyx is a promising new way to discover security vulnerabilities in Ruby-on-Rails web applications.

3.10 Languages as Libraries

Sam Tobin-Hochstadt (Northeastern University, Boston, US)

Joint work of Sam Tobin-Hochstadt, Robby Findler, Vincent St-Amour, Ryan Culpepper, Eli Barzilay, Matthew Flatt, and Matthias Felleisen

Programming language design benefits from constructs for extending the syntax and semantics of a host language. While C's string-based macros empower programmers to introduce notational shorthands, the parser-level macros of Lisp encourage experimentation with domain-specific languages. The Scheme programming language improves on Lisp with macros that respect lexical scope.

The design of Racket—a descendant of Scheme—goes even further with the introduction of a full-fledged interface to the static semantics of the language. A Racket extension programmer can thus add constructs that are indistinguishable from "native" notation, large and complex embedded domain-specific languages, and even optimizing transformations for the compiler backend. This power to experiment with language design has been used to create a series of sub-languages for programming with first-class classes and modules, numerous languages for implementing the Racket system, and the creation of a complete and fully integrated typed sister language to Racket's untyped base language.

3.11 Virtual Values for Language Extension

Cormac Flanagan (University of California, Santa Cruz, US)

Joint work of Thomas H.Austin, Tim Disney, and Cormac Flanagan

This paper focuses on extensibility, the ability of a programmer using a particular language to extend the expressiveness of that language. This paper explores how to provide an interesting notion of extensibility by virtualizing the interface between code and data. A virtual value is a special value that supports behavioral intercession. When a primitive operation is applied to a virtual value, it invokes a trap on that virtual value. A virtual value contains multiple traps, each of which is a user-defined function that describes how that operation should behave on that value.

This paper formalizes the semantics of virtual values, and shows how they enable the definition of a variety of language extensions, including additional numeric types; delayed evaluation; taint tracking; contracts; revokable membranes; and units of measure. We report on our experience implementing virtual values for Javascript within an extension for the Firefox browser.

3.12 Sandboxing Untrusted JavaScript

Ankur Taly (Stanford University, US)

Joint work of Ankur Taly, John C. Mitchell, Sergio Maffeis, Ulfar Erlingsson, Mark S. Miller, and Jasvir Nagra

Most websites today incorporate untrusted JavaScript content in the form of advertisements, maps and social networking gadgets. Untrusted JavaScript, if embedded directly, has complete access to the page's Document Object Model(DOM) and can therefore steal cookies, navigate the page, maliciously alter the page or cause other harm. In order to combat the above threat, many websites use language-based mechanisms for restricting untrusted JavaScript. Popular examples of such mechanisms are Facebook FBJS, Yahoo! ADSafe and Google Caja. In this talk, I will rigorously define the security goals of such sandboxing mechanisms and then develop principled techniques for designing and analyzing them. I will back the techniques with rigorous guarantees established using an operational semantics for JavaScript. I will also present security vulnerabilities in Facebook FBJS and Yahoo! ADSafe found during the course of this work and principled approaches to fixing those vulnerabilities. The talk will span JavaScript based on 3rd edition of the ECMA262 specification and also the recently released "strict mode" of JavaScript based on 5th edition of the ECMA262 specification.

3.13 ADsafety: Type-based Verification of JavaScript Sandboxing

Joe Politz (Brown University, Providence, US)

Joint work of Joe Gibbs Politz, Arjun Guha, Spirodon Aristides Eliopolous, and Shriram Krishnamurthi

Web sites routinely incorporate JavaScript programs from several sources into a single page. These sources must be protected from one another, which requires robust sandboxing. The many entry-points of sandboxes and the subtleties of JavaScript demand robust verification of the actual sandbox source. We use a novel type system for JavaScript to encode and verify sandboxing properties. The resulting verifier is lightweight and efficient, and operates on actual source. We demonstrate the effectiveness of our technique by applying it to ADsafe, which revealed several bugs and other weaknesses.

3.14 Integrating Typed and Untyped Code in a Scripting Language

Francesco Zappa Nardelli (Inria, Paris-Rocquencourt, FR)

Many large software systems originate from untyped scripting language code. While good for initial development, the lack of static type annotations can impact code-quality and performance in the long run. We present an approach for integrating untyped code and typed code in the same system to allow an initial prototype to smoothly evolve into an efficient and robust program. We introduce like types, a novel intermediate point between dynamic and static typing. Occurrences of like types variables are checked statically within their scope

but, as they may be bound to dynamic values, their usage is checked dynamically. Thus like types provide some of the benefits of static typing without decreasing the expressiveness of the language.

3.15 Using Contracts to Connect Different Scripting Languages

Kathryn E. Gray (University of Cambridge, GB)

Scripting languages are frequently combined with statically-typed languages, potentially running on different virtual machines. Conventional techniques for writing multi-language programs entail manually inserting data conversions, inter-machine communication, and dynamic checks, which can introduce subtle errors. My previous technique allows values to pass seamlessly from one language to another—for languages with similar dynamic semantics on the same VM. However, with scripting languages these criteria may not be met. So, this talk introduces a framework that supports languages with different runtime systems and semantics, while maintaining type-safety and a free exchange of values.

3.16 Blame for All

Philip Wadler (University of Edinburgh, GB)

Joint work of Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler

[Appeared in POPL 2009] Several programming languages are beginning to integrate static and dynamic typing, including Racket (formerly PLT Scheme), Perl 6, and C# 4.0, and the research languages Sage (Gronski, Knowles, Tomb, Freund, and Flanagan, 2006) and Thorn (Wrigstad, Eugster, Field, Nystrom, and Vitek, 2009). However, an important open question remains, which is how to add parametric polymorphism to languages that combine static and dynamic typing. We present a system that permits a value of dynamic type to be cast to a polymorphic type and vice versa, with relational parametricity enforced by a kind of dynamic selaing along the line proposed by Matthews and Ahmed (2008) and Neis, Dreyer, and Rossberg (2009). Our system includes a notion of blame, which allows us to show that when casting between a more-precise type and a less-precise type, any failure are due to the less-precisely-typed portion of the program. We also show that a cast from a subtype to its supertype cannot fail.

3.17 Temporal Higher-order Contracts

Cormac Flanagan (University of California, Santa Cruz, US)

Joint work of Tim Disney, Jay McCarthy, and Cormac Flanagan

Behavioral contracts are embraced by software engineers because they document module interfaces, detect interface violations, and help identify faulty modules (packages, classes, functions, etc). This paper extends prior higher-order contract systems to also ex- press and enforce temporal properties, which are common in soft- ware systems with imperative state, but which are mostly left implicit or are at best informally specified. The paper presents both a programmatic contract API as well as a temporal contract language, and reports on experience and performance results from implementing these contracts in Racket.

Our development formalizes module behavior as a trace of events such as function calls and returns. Our contract system provides both non-interference (where contracts cannot influence correct executions) and also a notion of completeness (where contracts can enforce any decidable, prefix-closed predicate on event traces).

3.18 A Racket Contract Example

Robert Bruce Findler (Northwestern University, Evanston, US)

The following contract is an example contract that illustrates why earlier (lax/picky) interpretations of dependent contracts are wrong. See also "Correct Blame for Contracts: No More Scapegoating" in POPL 2011.

```
#lang racket

(provide (contract-out [deriv/c contract?]))
(require (planet cce/fasttest:3/random))

(define n 10)
(define δ 0.01)

(define deriv/c
 (->i ([f (-> real? real?)]
       [ε (and/c real? positive?)])
      (fp (-> real? real?))
      \#:post
      (f fp ε)
      (for/and ([i (in-range 0 n)])
        (define x (random-number))
        (define slope
          (/ (- (f (+ x ε))
                (f (- x ε)))
             (* 2 ε)))
        (<= (abs (- slope (fp x))) δ))))
```

3.19 Dynamic Inference of Static Types for Ruby

Michael Hicks (University of Maryland, College Park, US)

Joint work of Michael Hicks, David An, Jeff Foster, and Avik Chaudhuri

Ruby is a dynamically typed scripting language in the tradition of Smalltalk. We have designed a type system for Ruby and a static type inference algorithm that we have applied to Ruby scripts and libraries. While a useful exercise, we found static type inference extremely difficult to develop: dynamic languages are typically complex, poorly specified, and include features, such as eval and reflection, that are hard to analyze.

In response, we developed constraint-based dynamic type inference, a technique that infers static types based on dynamic program executions. In our approach, we wrap each run-time value to associate it with a type variable, and the wrapper generates constraints on this type variable when the wrapped value is used. This technique avoids many of the often overly conservative approximations of static tools, as constraints are generated based on how values are used during actual program runs. Using wrappers is also easy to implement, since we need only write a constraint resolution algorithm and a transformation to introduce the wrappers. The best part is that we can eat our cake, too: our algorithm will infer sound types as long as it observes every path through each method body-note that the number of such paths may be dramatically smaller than the number of paths through the program as a whole.

We have developed Rubydust, an implementation of our algorithm for Ruby. Rubydust takes advantage of Ruby's dynamic features to implement wrappers as a language library. We applied Rubydust to a number of small programs and found it to be both easy to use and useful: Rubydust discovered 1 real type error, and all other inferred types were correct and readable.

3.20 Nested Refinements: A Logic for Duck Typing

Ravi Chugh (University of California, San Diego, US)

Joint work of Ravi Chugh, Pat Rondon, and Ranjit Jhala

Programs written in dynamic languages make heavy use of features—run-time type tests, value-indexed dictionaries, polymorphism, and higher-order functions—that are beyond the reach of type systems that employ either purely syntactic or purely semantic reasoning. We present a core calculus, System D, that merges these two modes of reasoning into a single powerful mechanism of nested refinement types wherein the typing relation is itself a predicate in the refinement logic. System D coordinates SMT-based logical implication and syntactic subtyping to automatically typecheck sophisticated dynamic language programs. By coupling nested refinements with McCarthy's theory of finite maps, System D can precisely reason about the interaction of higher-order functions, polymorphism, and dictionaries. We also discuss extensions to support imperative updates and inheritance, features commonly found in real-world dynamic languages.

3.21 The Ciao Assertions Model

Manuel Hermenegildo (IMDEA Software, Madrid, ES)

Joint work of M.V. Hermenegildo, M. Carro, P. López-García, J. Morales, F. Bueno, G. Puebla, R. Haemmerlé

We provide a brief overview (and demo!) of Ciao, emphasizing some of the novel aspects and motivations behind its design and implementation.

Ciao is built in layers over a kernel, which is designed to be extensible in a powerful, modular way. Using these facilities, Ciao provides the programmer with a large number of useful features from different programming paradigms and styles. All such features are in libraries, so that the use of each of the features (including those of logic and constraint programming) can be turned on and off at will for each program module. Thus, a given module may be using, e.g., higher order functions and constraints, while another module may be using assignment, predicates, meta-programming, and concurrency. The module system and the extension mechanism together allow user-level design and implementation of powerful extensions and domain specific languages.

Another important objective of Ciao as a "scripting language"—on which the talk and demo concentrate—is to offer the best of the dynamic and static language approaches, i.e., providing the flexibility of dynamic languages, but with guaranteed safety and efficiency. Important elements to this end are the Ciao assertion language and its preprocessor. The assertion language allows expressing many kinds of program properties (ranging from, e.g., moded types to resource consumption), as well as tests and documentation. The preprocessor is capable of statically finding violations of these properties or verifying that programs comply with them, and issuing certificates of this compliance, and also generating run-time tests for (parts of) specifications with which compliance cannot be resolved at compile-time. The compiler performs many types of optimizations (including automatic parallelization), producing code that is highly competitive with other dynamic languages or, with the (experimental) optimizing compiler, even that of static languages, all while retaining the flexibility and interactive development of a dynamic language.

3.22 Occurrence Typing

Sam Tobin-Hochstadt (Northeastern University, Boston, US)

Joint work of Sam Tobin-Hochstadt, Vincent St-Amour, and Matthias Felleisen

Ad-hoc, untagged unions are pervasive in scripting languages. However, traditional type systems do not handle unions well. In this talk, I describe occurrence typing, which provides an effective elimination rule for union types, and enables the type checking of idiomatic scripting language programs. I also describe a surprising application to numeric type checking.

3.23 Gradual Typing Roundup

Jeremy G. Siek (University of Colorado, US)

Gradual typing is an approach for integrating static and dynamic typing within the same language. Since it's introduction 5 years ago, many challenges have been overcome, such as how to efficiently represent higher-order casts and how to integrate gradual typing with other features such as objects and generics. This talk gives an example-based survey of the progress in gradual typing and discusses the remaining challenges, with some hints at solutions to some of them.

3.24 (Towards) Gradual Typing for Java

Atsushi Igarashi (Kyoto University, JP)

Joint work of Atsushi Igarashi and Lintaro Ina

We have presented our recent work on extending Java with gradual typing. The main focus is on the interaction between type "dynamic" and generic types. We have also discussed how our design constraint that proper Java code should compile to (almost) the same bytecode as javac affected the language feature design and implementation.

3.25 Combining Types and Flow Analysis

Arjun Guha (Brown University, Providence, US)

Joint work of Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Programs written in scripting languages employ idioms that confound conventional type systems. In this talk, we highlight one important set of related idioms: the use of local control and state to reason informally about types. To address these idioms, we formalize run-time tags and their relationship to types, and use these to present a novel strategy to integrate typing with flow analysis in a modular way. We demonstrate that in our separation of typing and flow analysis, each component remains conventional, their composition is simple, but the result can handle these idioms better than either one alone.

3.26 Lively Webwerkstatt—A Self-sustaining Web-based Authoring Environment

Jens Lincke, Robert Hirschfeld, and Bastian Steinert (Hasso-Plattner-Institut, Potsdam, DE)

Webwerkstatt is an interactive and programmable wiki environment for experimenting with different approaches to End-user Web Development and their boundaries to the application kernel. It is based on the Lively Kernel and incorporates projects such as Lively Wiki (a Wiki of live objects built on an SVN repository) and Lively Fabrik (a dataflow-like GUI builder for Lively). For Webwerkstatt, we developed the context-oriented language extension ContextJS, to explore new concepts for expressing this boundary. Our current research focuses on prototypical scripting and interactive application construction.

– http://lively-kernel.org/
– http://lively-kernel.org/repository/webwerkstatt/webwerkstatt.xhtml

3.27 What Use for Macros / Compile-time Meta-programming?

Laurence Tratt (King's College, London, GB)

Is the oft-repeated idea that "all good languages have macros / CTMP" undeniably true? This short talk is intended to make us think about the consequences of this idea.

3.28 Experiences of Implementing a VM with RPython

Laurence Tratt (King's College, London, GB)

A report on preliminary work on implementing an RPython VM for Converge, which suggests that language designers of the future now have a practical route for making "fast enough" VMs in "fast enough" time. See also http://convergepl.org/

3.29 Meta-Tracing in the PyPy Project for Efficient Dynamic Languages

Carl Friedrich Bolz (Universität Düsseldorf, DE)

Joint work of Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo

Writing JIT-compilers for recent scripting languages is a hard problem due to their recent semantics. The PyPy project tries to help with that problem by providing a tracing JIT

that operates "one level down", i.e. below an interpreter. That way the JIT can be reused for a number of languages.

3.30 HipHop – A Synchronous Reactive Extension for Hop

Manuel Serrano (Inria, Sophia Antipolis, FR)

HOP is a SCHEME-based language and system to build rich multi-tier web applications. We present HIPHOP, a new language layer within HOP dedicated to request and event orchestration. HIPHOP follows the synchronous reactive model of the Esterel and ReactiveC languages, originally developed for embedded systems programming. It is based on synchronous concurrency and preemption primitives, which are known to be key components for the modular design of complex temporal behaviors. Although the language is concurrent, the generated code is purely sequential and thread-free; HIPHOP is translated to HOP for the server side and to straight JavaScript for the client side. With a music playing example, we show how to modularly build non-trivial orchestration code with HIPHOP.

3.31 A Possible End-User Scripting Environment for STEPS

Yoshiki Ohshima (Viepoints Research Institut, Glendale, US)

In this talk, a brief overview of the STEPS Project and a possible design and implementation of the end-user scripting environement are presented.

In the STEPS Project, we are set out to explore good abstractions and concise descriptions of the entire personal computing environment. Language execution engines, a graphics engine, a GUI framework and an application framework were created in this philosophy and a universal document editor is created.

However, an end-user scripting system is yet to be written. Drawing from the Functional Reactive Programming work, we are exploring reactive programming in more dynamic setting. A possible implementation of such an end-user scripting system is under development.

3.32 101companies:101 Ways of Building a Management System With Different Programming Technologies

Ralf Lämmel (Universität Koblenz-Landau, DE)

The open-source 101companies Project is concerned with aggregating, organizing, annotating, and analyzing a corpus of many implementations of a simple Human Resource Management System (the so-called 101companies System) such that the implementations leverage varying programming technologies and varying software languages dedicated to different technological

spaces. The specification of the 101companies System contains several optional features which implementations can choose to implement in the interest of demonstrating specific programming technologies or capabilities thereof. The 101companies Project helps understanding and comparing programming technologies in a manner as it is valuable for different stakeholders such as teachers, learners, developers, software technologists, and ontologists. In this paper, we present the following major aspects of the project: i) an emerging ontology of relevant entities and categories; ii) a list of stakeholders of the project; iii) a feature model of the 101companies System; iv) themes as a grouping concept for implementations of the system; v) the structured documentation of implementations.

3.33 A Scripting Language for Remote Communication

William R. Cook (University of Texas, Austin, US)

Joint work of William R. Cook, Eli Tilevitch, Ben Wiedermann, and Ali Ibrahim

Batches are a new approach to relational database access, remote procedure calls, and web services. Batching employs a simple scripting language to communicate work from a client to a server. Batches also have a new control flow construct, called a Remote Batch statement. A Remote Batch statement combines remote and local execution: all the remote code is executed in a single round-trip to the server, where all data sent to the server and results from the batch are communicated in bulk. Batches support remote blocks, iteration and conditionals, and local handling of remote exceptions. Batches are efficient even for fine-grained interfaces, eliminating the need for hand-optimized server interfaces.

Batch services also provide a simple and powerful interface to relational databases, with support for arbitrary nested queries and bulk updates. One important property of the system is that a single batch statement always generates a constant number of SQL queries, no matter how many nested loops are used.

3.34 Languages in Racket Demo

Matthew Flatt (University of Utah, Salt Lake City, US)

Racket provides a smooth path from syntactic abstraction, language extension, language implementation, and environment support for languages with or without S-expression notation. In this demonstration, we show how implement a little JavaScript-like language in about 200 lines of code (mostly a parser).

Participants

- Amal Ahmed
Northeastern Univ. – Boston, US
- Carl Friedrich Bolz
Universität Düsseldorf, DE
- Ravi Chugh
University of California – San Diego, US
- William R. Cook
University of Texas – Austin, US
- Theo D'Hondt
Vrije Universiteit Brussel, BE
- Matthias Felleisen
Northeastern University – Boston, US
- Robert Bruce Findler
Northwestern Univ. – Evanston, US
- Cormac Flanagan
University of California – Santa Cruz, US
- Matthew Flatt
University of Utah – Salt Lake City, US
- Jeffrey Foster
University of Maryland – College Park, US
- Andreas Gal
Mozilla – Mountain View, US
- Philippa Gardner
Imperial College London, GB
- Kathryn E. Gray
University of Cambridge, GB
- Arjun Guha
Brown Univ. – Providence, US
- Stefan Udo Hanenberg
Universität Duisburg-Essen, DE
- David Herman
Mozilla – Mountain View, US
- Manuel Hermenegildo
IMDEA Software – Madrid, ES
- Michael Hicks
University of Maryland – College Park, US
- Robert Hirschfeld
Hasso-Plattner-Institut – Potsdam, DE
- Atsushi Igarashi
Kyoto University, JP
- Shriram Krishnamurthi
Brown Univ. – Providence, US
- Ralf Lämmel
Universität Koblenz-Landau, DE
- Benjamin Lerner
University of Washington – Seattle, US
- Jens Lincke
Hasso-Plattner-Institut – Potsdam, DE
- Hidehiko Masuhara
University of Tokyo, JP
- Mark S. Miller
Sunnyvale, US
- Floreal Morandat
Purdue University, US
- Oscar M. Nierstrasz
Universität Bern, CH
- Nathaniel Nystrom
Universität Lugano, CH
- Yoshiki Ohshima
Viepoints Research Institut – Glendale, US
- Joe Politz
Brown Univ. – Providence, US
- Gregor Richards
Purdue University, US
- Manuel Serrano
Inria – Sophia Antipolis, FR
- Jeremy G. Siek
University of Colorado, US
- Bastian Steinert
Hasso-Plattner-Institut – Potsdam, DE
- Ankur Taly
Stanford University, US
- Eric Tanter
Univ. of Chile – Santiago, CL
- Sam Tobin-Hochstadt
Northeastern University – Boston, US
- Laurence Tratt
King's College – London, GB
- Herman Venter
Microsoft Res. – Redmond, US
- Jan Vitek
Purdue University, US
- Philip Wadler
University of Edinburgh, GB
- Francesco Zappa Nardelli
Inria – Paris-Rocquencourt, FR