B1.c Extended Synopsis

1. Problem Statement

Cloud computing is changing our perception of computing: The Internet is becoming both the computer and the software: (a) vast data centers and computing power are available via the Internet ("infrastructure as a service"), (b) software is available via the Internet as a service, often in a multi-tenant mode ("software as a service"). The promise of unlimited processing/storage power has fostered data intensive and reactive applications processing big amounts of data from heterogenous sources scattered over the cloud and reacting to events happening across the cloud. Software services must be both standard components to pay off for their provider and highly configurable and customizable to serve the competitive needs of multiple tenants. Developing such applications is challenging using current programming technology.

First, existing abstractions are laid out to mostly process individual values. This model forces programmers of reactive and data intensive computations to explicitly track the data and keep tabs on events across a cloud and programmatically correlate them. This introduces accidental complexity and channels programmers attention and effort away from the relevant program logic building on top of big data processing. Making the complexity of data intensive and reactive software manageable requires abstractions to express high-level correlations between data/events, freeing the programmer from the task of explicitly tracking them. Ideally, means of abstraction, e.g., object-oriented subtype polymorphism, should be applicable to these abstractions, too.

Second, existing programming abstractions fail to reconcile software reuse and extensibility at the level of large-scale software services. Object-oriented programming (OOP) reconciles extensibility and reuse of individual classes via inheritance, overriding, and subtype polymorphism. However, these features do not scale up to units of deployment and aggregation in a large-scale assembly process such as packages or namespaces. These deficiencies make it difficult to provide service variations to accommodate specific needs of (groups of) tenants and to enable several tenants to share a service implementation with different configurations [4]. To work around these deficiencies, dependency injection frameworks [37, 43, 18, 23] have emerged that provide substantial infrastructure for naming and packing classes, for describing modules and making them identifiable, for using class loading to resolve module dependencies, for call-backs to intercept the inter-module communication by the framework, etc., significantly contributing to software complexity.

2. Objectives

The goal of PACE is to design, implement, and validate a programming model that cohesively integrates abstractions for reactive and data intensive computations with large-scale modularity concepts to radically improve the development and quality of applications in cloud environments.

Unified Abstractions for Reactive and Data Intensive Computations. PACE will unify and further develop separated threads of existing work on: (a) reactive behavior (RB), (b) language integrated queries (LIQ), and (c) distributed data-parallel computing (DDPC). The RB thread includes work on functional reactive programming [6, 30, 27], aspect-oriented programming [24, 8, 1, 38], language integrated event processing [13, 22, 17], and coordination of concurrent processes [3, 1, 32, 20, 19, 13]. Each class of approaches addresses a different aspect of reactive behavior, and none

of them handles reactive behavior over mutable object graphs. The most notable LIQ approach is .Net's LINQ framework [28][21] supporting uniform access over in-memory data collections, relational databases, and XML stores. The need for high-level abstractions for DDPC has lead to various frameworks and domain-specific languages [5, 36, 39, 21]. LIQ and DDPC threads focus on different aspects of data processing (bridging between data models and facilitating efficient processing of data); none makes data correlation a first-class concept that is e.g., subject to inheritance, and subtype polymorphism.

Our objective is to address the specific problems in each thread not individually, but by a unifying approach based on identifying the essential underlying abstractions. Apart from the general argument that uniformity enables better abstractions and reduces the number of concepts to reason about, there are specific theoretical, conceptual, and technical arguments for a unified approach. From a theoretical perspective, category theory suggests dualities between all three areas. Meijer and Biermann [29] have observed that relational data (SQL-like) and object graphs (noSQL-like) are duals in the category theory sense. Duality seems to relate RB and LIQ concepts, too [7]. From a conceptual perspective, approaches to RB, DDPC, and LIQ are similar in that all three try to replace a programming model organized around individual passive values by one organized around changing data sets (events involved in reactive behavior, in-memory collections, or data scattered over the Internet). From a technical perspective, unification paves the road to a common core set of efficient implementation techniques. Viewed from a unifying perspective, implementation techniques used in current approaches to RB, LIQ, and PDDB reveal several commonalities. Both incremental view maintenance and reactive behavior are based on building data-flow graphs to incrementally propagate changes. Just as LINQ uses reified query tree expressions for query optimizations, FlumeJava [5] also builds a dependency graph of operations involved in a data processing pipeline for deriving optimized evaluation plans.

The key novel contribution of PACE is a unified object-oriented language model that has data/event correlations as a primitive element and defines means of composition and abstraction for them. By being a first-class element of the language, data/event correlations become extensible by inheritance, late binding and subtype polymorphism. Furthermore, the model provides a set of abstractions on which correlations operate that enable polymorphism with respect to what is processed and how the processing is done; yet, different instantiations of these abstractions will account for the specific semantics of event or data processing. Armed with such a language model, programmers could turn their attention away from micromanaging data/events to taking advantage of what the cloud offers. The resulting applications become easier to understand/evolve and more amenable to automated reasoning and sophisticated optimization techniques. A unified language model for reactive and data intensive computations as sketched here is visionary and has the potential to trigger substantial new research.

Large-Scale Modularity. PACE will deliver language concepts for large-scale modularity, reuse, and extensibility to enable polymorphic software services, thereby building on and extending work of the PI on CaesarJ [31, 2][22].

CaesarJ uniformly encodes small-scale and large-scale modules by classes with nested classes. Simi-

[21] There are also some preliminary efforts to support language integrated queries in Java [44] and Scala [42, 14].

[22] CaesarJ belongs to a class of approaches to advanced modularity in OO languages that uniformity encode small-scale and large-scale modules in classes [10, 31, 2, 33, 4]. Details of other approaches are presented in the proposal.

larly, module interfaces are encoded in interfaces with nested interfaces. Instances of module classes encapsulate the implementation of a whole class graph (class family), enabling family polymorphism [10, 11]; a special form of dependent types [12] is used for type-safe family polymorphism. Large-scale module extensibility is enabled by virtual classes[23] [26], and propagating mixin composition [40, 9, 41]. Intermodule dependencies are expressed via collaboration interfaces that in addition to what is provided by a module also declare what the module expects from other modules. Module implementations realize the provided part; module bindings realize the expected part[24]. Consider e.g., a graphical editor module involving abstractions for nodes and connectors; its provided interface covers the "view"[25] aspects of these abstractions; the expected part covers the "model" aspects. Several implementations of the editor are conceivable, e.g., with support for accelerator keys, with multi-language support, etc.; likewise, there can be several bindings to display different data types, e.g., nodes in a WLAN, participants in a social network, etc. Implementations and binding of the same interface are automatically composable via propagating mixin composition; one can reuse a service implementation with any binding of that service's interface and vice versa.

These features make CaesarJ's concepts a great starting point towards language support for "software-as-a-service". PACE will advance the current state of the art in three important ways.

First, virtual classes will be generalized. The latter can modularly express only one dimension of class variability, represented by the type hierarchy of the enclosing object. There are, however, scenarios where a class' definition may vary along several dimensions, e.g., the definition of nodes of a graphical editor depends on both the editor type and the data types displayed by the node. A generalization of virtual classes are dependent classes [16]: Variation axes of a class are expressed by parameters rather than by nesting. Preliminary investigations [15] indicate that this idea has great potential for enabling highly polymorphic definitions of functionality, particularly of the kind needed in the area of reactive and data intensive applications. In PACE, this idea will be realized in a comprehensive research effort answering several open questions: How would the design of a calculus that formalizes the idea look like? Does it make sense to have both dependent and virtual classes? Can a generalization of dependent classes subsume generics? How can more dynamic variations be supported while retaining type safety? And, many more.[26]

Second, PACE will deliver a concept for intermodule dependency management and composition that avoids static dependency resolution. Like ML functors [25] and Newspeak modules [4], Caesar's family classes abstract over their dependencies via parameterization by the collaboration interface. Unlike in Newspeak, composition does not happen via parameter passing, but by propagating mixin composition, thus, it is static like in ML. However, dynamic service composition is an important feature in cloud environments. The challenge will be to reconcile the power of a parameter-passing style mechanism for module dependency resolution with static typing.

The Overall Vision. The overall objective is a a seamless integration of large-scale modularity concepts with abstractions for reactive and data intensive computations yielding great synergies for both sides. This integration will make reactive and data intensive software subject to large-scale reusability and incremental extensibility, e.g., it will allow programmers to encode the behavior

[23] Like methods, nested classes can be overridden in subclasses, hence they are called virtual classes.

[24] An implementation of the expected part is mostly used to map service abstractions to domain abstractions in a particular usage context of the service, hence the name.

[25] In the sense of the Model-View-Controler (MVC) pattern.

[26] More details about the research questions related to dependent classes to be addressed are given in the proposal.

of large-scale components as state machines and enable extensibility at this level of abstraction. In turn, the integration will decouple service bindings from the specifics of the binding context. CaesarJ uses aspect-oriented pointcut-advice mechanisms to embed bindings in the control and data flow of a service's usage context, which may introduce coupling [38]; this would be reduced by proper abstractions for reactive behavior.

3. Work Plan

Scala [35] will be used as our workbench, since it offers features relevant for PACE, e.g., higher-oder functions and advanced typing, and support for embedding new concepts without changing the compiler. Moreover, it is unique in having vibrant research and industrial communities. A collaboration with Prof. Dr. Martin Odersky and the Scala team at EPFL is planned. We have informally agreed on the goals of this proposal to avoid duplication of research efforts and to coordinate our efforts with the work on the Doppler project[27]. The planned cooperation offers the opportunity to integrate the results of PACE into the development version of Scala. Our goal is not to "stuff" Scala with concepts, but to introduce concepts that generalize over existing ones, thus any new concept should ideally make some existing one obsolete.

The research will be organized in the following units summarized in Figure 1.

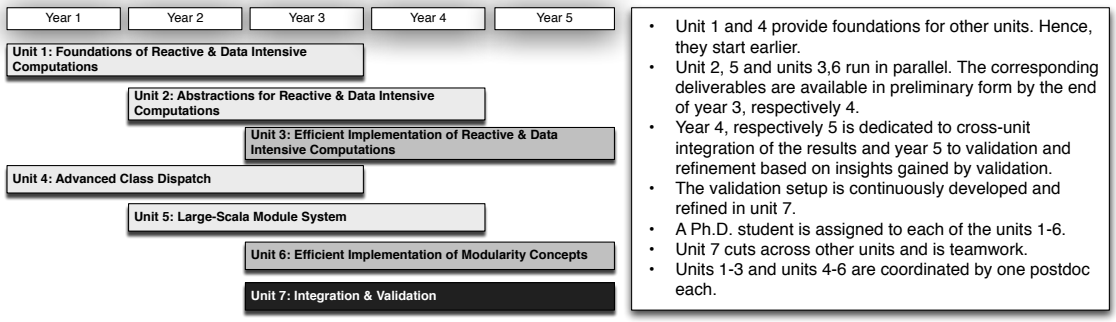

Figure 1: Work Units

Unit 1: Foundations of Abstractions for Reactive and Data Intensive Computations. This unit will develop a common theoretical foundation for language integrated queries, abstractions for data-parallel pipelines, and reactive programming. It will explore the suitability of structures from category theory as a foundation for the abstractions for reactive and data intensive computations. The key challenge will be the application of these structures to an OO setting, featuring subtype polymorphism; so far they have been used in a functional setting, featuring parametric polymorphism.

Unit 2: Abstractions for Reactive and Data Intensive Computations. The first key challenge is support for declaring arbitrary computations as reactive, which causes their results to be refreshed whenever data changes that participated in computing them. A second challenge concerns the design of a unified query language that enables abstraction over details of the data being processed

[27] ERC Advanced Grant awarded to Martin Odersky

and over technical aspects of computations. The third key challenge concerns the design of a library of polymorphic data types that have multiple definitions, each with a specialized implementation of the procedural interface encoding specific processing strategies.

Unit 3: Efficient Implementation of Abstractions for Reactive and Data Intensive Computations. This unit will develop (a) a common set of query optimization techniques that apply across event and data queries, and (b) a common push-based dependency tracking infrastructure for reactive data, event/handler dependencies, and propagation of changes incrementally to cached query results. This unit involves interesting interactions of techniques from databases and programming languages.

Unit 4: Advanced Class Dispatch. Advanced class dispatch will be formally defined, departing from previous formalizations of virtual classes [16] and the core of Scala [34]. Based on insights gained in previous work, other ways to represent types will be explored, e.g., as sets of constraints. Encoding generics as dependent classes and advancing dispatch and class composition semantics are further topics that will be investigated.

Unit 5: Large-Scale Module System. This unit is concerned with designing a parametric module system inspired by that of Newspeak. The key challenges are (a) to come up with a unified design bringing together parameterized modules and parameterized classes (dependent classes) in a meaningful way, and (b) to ensure static type safeness in the presence of increased flexibility.

Unit 6: Efficient Implementation of Modularity Concepts. This unit will integrate results of unit 5 unit 6 into a practically usable language by studying and developing efficient implementation techniques for the proposed modularity concepts. An appropriate mapping of the new concepts to Scala concepts, such as virtual types and traits, will be designed. Virtual machine techniques will also be considered.

Unit 7: Integration and Validation. This unit designs and conducts empirical studies to answer questions such as: (1) Do dedicated language abstractions reduce the complexity and improve overall design quality of applications in cloud environments compared to mainstream programming models? (2) Does the unified language design generalize over existing abstractions, yield better designs, and is less complex? (3) Do dedicated abstractions positively/negatively affect the efficiency of applications in a cloud environment? Some of the case studies will be derived from real cases in the context of Business ByDesign, the service platform of SAP. I am advising a Ph.D. student funded by SAP, who is working on improving module and extensibility mechanisms of the Business ByDesign platform. Demonstrators for these case studies will be built for experimental cloud environments for quantitative evaluations.

1.1 Summary

PACE is an exciting endeavor that has the potential to create a new programming paradigm that will revolutionize the way we develop software for a novel computing model. Of course, as any project of this kind, PACE will not deliver a silver bullet, but rather foundations to build upon and to develop further. This is a very ambitious project, where many parts have to work well individually and in concert, in order to reach the objectives. Yet, my significant achievements in programming language design in the last 10 years and the planned cooperation with the Scala team at EPFL make PACE feasible.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. *SIGPLAN Not.*, 40:345–364, 2005.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I, Lecture Notes in Computer Science, Volume 3880/2006*, pages 135–173, 2006.

[3] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

[4] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *Proceedings of European Conference on Object-oriented Programming*, ECOOP'10, pages 405–428. Springer-Verlag, 2010.

[5] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. *SIGPLAN Not.*, 45:363–375, 2010.

[6] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *European Symposium on Programming*, pages 294–308, 2006.

[7] Smet DeBart. Keynote, European Conference on Object-Oriented Programming, ECOOP 2011. 2011.ecoop.org/.

[8] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings International Conference on Aspect-Oriented Software Development*, AOSD '04, pages 141–150. ACM, 2004.

[9] Erik Ernst. Propagating Class and Method Combination. In *Proceedings of European Conference on Object-Oriented Programming*, ECOOP'99, pages 67–91. Springer-Verlag, 1999.

[10] Erik Ernst. Family Polymorphism. In *Proceedings of European Conference on Object-Oriented Programming*, ECOOP '01, pages 303–326. Springer-Verlag, 2001.

[11] Erik Ernst. Higher-Order Hierarchies. In *Proceedings of European Conference on Object-oriented Programming*, ECOOP'03, pages 303–328. Springer-Verlag, 2003.

[12] Erik Ernst, Klaus Ostermann, and William R. Cook. A Virtual Class Calculus. *SIGPLAN Notices*, 41:270–282, 2006.

[13] Patrick Th. Eugster and K.R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *Proceedings of European Conference on Object-oriented Programming*, ECOOP'09, pages 570–594. Springer-Verlag, 2009.

[14] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with Database Query Capability. *Journal of Object Technology*, 9(4):45–68, 2010.

[15] Vaidas Gasiunas. *Advanced Object-Oriented Language Mechanisms for Variability Management*. PhD thesis, TU Darmstadt, Germany, Dezember 2010.

[16] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent Classes. *SIGPLAN Notices*, 42:133–152, October 2007.

[17] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular Event-Driven Object Interactions in Scala. In *Proceedings of International Conference on Aspect-Oriented Software Development*, AOSD '11, pages 227–240. ACM, 2011.

[18] Guice. http://code.google.com/p/google-guice/.

[19] Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.*, 410(2-3), 2009.

[20] Philipp Haller and Tom van Cutsem. Implementing Joins using Extensible Pattern Matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science, Vol. 5052, pages 135–152. Springer-Verlag, 2008.

[21] Michael Isard and Yuan Yu. Distributed Data-parallel Computing Using a High-level Programming Language. In *Proceedings of SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 987–994. ACM, 2009.

[22] K.R. Jayaram and Patrick Th. Eugster. Scalable Efficient Composite Event Detection. In *Proceedings of International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science, Volume 6116, pages 168–182, 2010.

[23] Jigsaw Module System. http://openjdk.java.net/projects/jigsaw/.

[24] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer-Verlag, 2001.

[25] David MacQueen. Modules for Standard ML. In *Proceedings of ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207. ACM, 1984.

[26] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-oriented Programming. *SIGPLAN Notices*, 24:397–406, September 1989.

[27] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the Observer Pattern. Technical report, EPFL, 2010.

[28] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706. ACM, 2006.

[29] Erik Meijer and Gavin Bierman. A Co-relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 54:49–58, April 2011.

[30] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. *SIGPLAN Not.*, 44:1–20, 2009.

[31] Mira Mezini and Klaus Ostermann. Conquering Aspects with Caesar. In *Proceedings of International Conference on Aspect-oriented Software Development*, AOSD '03, pages 90–99. ACM, 2003.

[32] Angel Núñez and Jacques Noyé. An Event-based Coordination Model for Context-aware Applications. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science, Vol. 5052, pages 232–248. Springer-Verlag, 2008.

[33] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable Extensibility via Nested Inheritance. *SIGPLAN Notices*, 39:99–115, 2004.

[34] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of European Conference on Object-oriented Programming*, ECOOP '03, pages 201–224, 2003.

[35] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[36] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110. ACM, 2008.

[37] OSGi Service Platform Specification. http://www.osgi.org/Specifications/HomePage.

[38] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of European Conference on Object-Oriented Programming*, ECOOP'05, pages 214–240. Springer-Verlag, 2005.

[39] Parallel LINQ. http://msdn.microsoft.com/en-us/library/dd460688.aspx.

[40] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of European Conference on Object-Oriented Programming*, ECOOP'98, pages 550–570. Springer-Verlag, 1998.

[41] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11:215–255, April 2002.

[42] Daniel Spiewak and Tian Zhao. ScalaQL: Language-Integrated Database Queries for Scala. In *International Conference on Software Language Engineering, SLE*, volume 5969 of *LNCS*, pages 154–163. Springer, 2009.

[43] The Spring Framework. http://www.springsource.org.

[44] Darren Willis, David J. Pearce, and James Noble. Caching and Incrementalisation in the Java Query Language. In *Proceedings of ACM Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 1–18. ACM, 2008.