A Core Calculus for Static Latency Tracking with Placement Types

Tobias Reinhard
Technische Universität Darmstadt
Graduate Student, ACM-Nr: 3692899
reinhard@stud.tu-darmstadt.de

ABSTRACT

Developing efficient geo-distributed applications is challenging as programmers can easily introduce computations that entail high latency communication. We propose a language design which makes latency explicit and extracts type-level bounds for a computation's runtime latency. We present our initial steps with a core calculus that enables extracting provably correct latency bounds and outline future work.

1 INTRODUCTION

Developing efficient geo-distributed applications remains a challenging task. Efficiency is largely determined by latency caused by remote communication. Avoiding high-latency remote communication and exploiting locality is therefore imperative [15]. Distributed components are, however, often interconnected and local computations can trigger a chain of events causing high-latency remote computations [10, 11, 13]. Determining which local computations eventually lead to latency, introducing remote communication, often requires a global view which hinders modular development of geo-distributed software. Also, the exact location where a remote computation is placed, matters. Communication among servers in a single data center, for instance, is much faster (under 2ms) than communication between geo-distributed data centers possibly located on different continents (over 100ms) [3].

We build on the idea of making locations and latency explicit [14] and adopt the approach that a computation's location and its entailed latency become part of its type. The type system can infer an upper bound on a computation's actual latency and can reject code containing wrong assumptions, e.g., on the latency caused by a method invocation. As a method signature already describes the latency its invocation entails, no global view is required anymore and code becomes more modular.

In this work, we present our work on formalizing the type system in a core calculus λ^{lat}, on extracting latency bounds, and on proving their correctness. We outline ongoing work about further extending the formalization and about enabling latency-saving refactorings. Finally, we discuss how we plan to evaluate this research line.

Supervisors: Guido Salvaneschi (Technische Universität Darmstadt), Pascal Weisenburger (Technische Universität Darmstadt).

2 A CALCULUS FOR LATENCY

In the following we present λ^{lat} as well as the ideas behind its correctness proof. The goal of λ^{lat} is to track the location (called a *peer*), on which a computation is run, and the latency it causes in the type system. In this calculus, a computation's latency refers to the weighted number of remote messages sent during the computation. The time a message needs to reach its recipient depends on the involved peers. For a fixed set of peers, e.g., found in geo-distributed data centers, we can assume fixed locations, and thus, known latency values, e.g., known from monitoring. We therefore use a function $\mathcal{L} : \mathbb{P} \times \mathbb{P} \to \mathbb{N}$ to assign weights (i.e., latency approximation) $\mathcal{L}(P, P')$ to the messages sent from peer P to P'.

Dynamic Semantics. λ^{lat} is an extension of the typed lambda calculus [2] where every computation is placed on a specified location. Peer types P and peer instances p specify type-level and runtime locations, respectively. Types are augmented by sizes and latency bounds. We use a fragment of Heyting arithmetic [9] containing 0, S, $+$, $\dot{-}$ and $\cdot$ to define those and prove arithmetic properties.

The small step reduction relation $\overset{\mathcal{I}}{\rightsquigarrow}$ describes a reduction step on a set of peer instances $\mathcal{I}$. Locations and latency are explicit in the reduction semantics. Every intermediate result as well as the end result of a term's reduction is annotated by its location and the latency its reduction has caused. A peer evaluation context $(\langle t \rangle_{\mathcal{I}}, [l])$ describes a term t to be evaluated on a set of peer instances $\mathcal{I}$ where l is the latency that has been caused during the reduction so far. A reduction step $(\langle t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t' \rangle_{\mathcal{I}}, [l'])$ expresses that term t is reduced on the peer instances $\mathcal{I}$ to t' and that the tracked latency increases from l to l'. Local reduction steps are standard and leave the tracked latency unchanged. Also, no latency-decreasing steps exist. Hence, we have $l \leq l'$. However, every reduction step involving a message sent from a peer P to P' increases the tracked runtime latency by the weight $\mathcal{L}(P, P')$

Consider the evaluation of the term $\text{get}\, p'.v$ on a set $\mathcal{I}^P$ of P-instances. The expression requests a value v from p' and can be reduced by sending the request to p', increasing the latency by $\mathcal{L}(P, P')$. Hence, the reduction step is $(\langle \text{get}\, p'.v \rangle_{\mathcal{I}^P}, [l]) \overset{\mathcal{I}^P}{\rightsquigarrow} (\langle (\langle v \rangle_{\{p'\}}, [0]) \rangle_{\mathcal{I}^P}, [l + \mathcal{L}(P, P')])$. A remote evaluation $(\langle v \rangle_{\{p'\}}, [0])$ starts with a remote latency of 0. When the result is transmitted, the remote latency is added to the local latency. Since we assume v to be a value, it cannot be reduced any further and the next step is sending v from p' to $\mathcal{I}^P$. Runtime latency thereby increases to $l + \mathcal{L}(P, P') + \mathcal{L}(P', P)$.

Static Semantics. Assigning types to well-formed terms t ensures that there is a sequence of reduction steps towards a value $(\langle v \rangle_{\mathcal{I}}, [l])$ and that v belongs to that type. Our approach lifts the latency of every reduction step to the type level. Additionally we ensure termination of recursive functions by employing sized types [1] and only allowing size-decreasing recursion. A type is a triple $(B, [s], [l])$.

Conference'17, July 2017, Washington, DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

B is a basic type like Unit determining the kind of value a term reduces to, s and l are arithmetic terms representing the value's size and an upper bound on the latency caused during reduction. $[s]$ and $[l]$ denote the equivalence classes of terms provably equal to s and l, respectively, in Heyting arithmetic.

Type-level Latency. Considering the term $\mathsf{get}\, p'.t$ (similiar to the example above but without assuming p' and t to be values) and abstracting over concrete peer instances: Evaluating this term means (i) evaluating p' on the current peer P, (ii) sending a request for t to the remote peer P', (iii) waiting for its evaluation and (iv) P' sending the result to P. We lift the runtime latency $l_{p'} + \mathcal{L}(P, P') + l_t + \mathcal{L}(P', P)$ to the type level, as typing rule T-Get shows:

$$\frac{P \leftrightarrow P' \qquad \Delta;\Gamma;\Lambda;P \vdash p' : (P', [0], [l_{p'}]) \qquad \Delta;\emptyset;\emptyset;P' \vdash t : (B, [s], [l_t])}{\Delta;\Gamma;\Lambda;P \vdash \mathsf{get}\, p'.t : (\mathsf{Option}\,(B, [s]), [0], [l_{p'} + \mathcal{L}(P, P') + l_t + \mathcal{L}(P', P)])} \quad \text{(T-Get)}$$

Latency Bounds. Analyzing a term's structure is not always sufficient to compute its exact runtime latency. In general, terms can have multiple reduction sequences resulting in different runtime latencies. We therefore consider type-level latency as an upper bound on all possible runtime latencies, the typed term can reduce to. For instance, considering the term $\mathsf{if}\ t_c\ \{t_t\}\ \{t_f\}$: In any case, the condition t_c is evaluated and depending on the result also one of the branches t_t and t_f. As shown by rule T-If, we extract an upper bound by taking the maximum over both branches' latency:

$$\frac{\Delta;\Gamma;\Lambda;P \vdash t_c : (\mathsf{Boolean}, [0], [l_c]) \qquad \Delta;\Gamma;\Lambda;P \vdash t_t : (B, [s], [l_t]) \qquad \Delta;\Gamma;\Lambda;P \vdash t_f : (B, [s], [l_f])}{\Delta;\Gamma;\Lambda;P \vdash \mathsf{if}\ t_c\ \{t_t\}\ \{t_f\} : (B, [s], [l_c + \max(l_t, l_f)])} \quad \text{(T-If)}$$

Size-dependent Functions. In the case of functions, the latency can depend on the input's size. Considering a list processing function f, where the processing of each element involves some latency l: For any list a of size s_a the latency of an application $f\ a$ is $s_a \cdot l$. In λ^{lat}, such a function's basic type has the form $\forall(s : \mathbb{N}) . (B, [s]) \to (B', [s'], [l'])$. It expresses that the function can handle arguments of type B and arbitrary size s and that every such argument is mapped to a value of type $(B', [s'], [l'])$ where variable s may occur free in s', l'. For instance, in the previous example, we get $l' = s \cdot l$.

Size-decreasing recursion. Recursion is a convenient way to define size-dependent functions. In λ^{lat}, the only way to express recursion is via a fixpoint operator. Our type system ensures that for every application to an argument of size s, the recursive step is taken on a smaller argument of size $s' < s$. Since sizes are finite, our type system ensures termination. Thus, application of the fixpoint operator preserves the correctness of extracted latency bounds.

Correctness. We have shown how our type system lifts runtime latency to the type level and extracts upper bounds for branching terms. We also showed how we can extract latency bounds for recursive function applications. Hence, we can prove the following:

Theorem 1 (Correctness of Type-level Latency Bounds). *Let Δ and Γ be typing environments for placed and local variables, respectively. Let Λ be a set of arithmetic assumptions and P a peer type, $\mathcal{I}$ a set of peer instances, t a term, v a value. Further, let B be a basic type, s a size and l_R, l_T latencies. Suppose $\Delta;\Gamma;\Lambda;P \vdash t : (B, [s], [l_T])$ and that there exists a reduction sequence for t to a value $(\langle v \rangle_{\mathcal{I}}, [l_R])$. Then $l_R \leq l_T$ holds.*

3 OUTLOOK

We are currently investigating whether the extracted type-level bounds are optimal regarding the worst-case runtime latency. This is particularly interesting for recursive functions where we need to check that an input exists that (i) causes the estimated maximal number of recursive steps and (ii) in each step meets the extracted latency bound. As message delay in distributed systems is non-deterministic, we plan to refine our approach by using probability distributions for the latency weights $\mathcal{L}(P, P')$ instead of natural numbers.

An important aspect to consider is to complement the (static) analysis provided by the type system with actual latency measurements collected via monitoring. We believe that the combination of both can provide correct feedback to the developers. To this end, we are working on a monitoring system that provides realistic estimations for latency and retrofits them in the type system using methods from continuous integration.

We are currently implementing a prototype of the language presented in [14] based on the type system of λ^{lat}. Eventually, we are going to implement type-based latency tracking in ScalaLoci [12], a multitier language whose type system keeps track of a computation's location similar to λ^{lat}.

Using ScalaLoci's extended type system, we are going to explore latency-saving refactorings. High-latency inducing computations often contain unnecessary remote communication. Relocating parts of the computation and only transmitting as few data as necessary helps to reduce latency. We believe that the combination of static location and latency information is sufficient to implement such refactorings.

We plan to evaluate the type system's usability with controlled experiments and case studies on applications involving multiple geo-distributed data centers. Using platforms like Amazon AWS, we plan to use real locations for the data centers [6] and to specify realistic latency weights $\mathcal{L}(P, P')$ for the connections.

4 RELATED WORK

This paper builds on our previous work presenting the design of a language which makes latency transparent to the programmer [14]. To the best of our knowledge, no previous work formally explores type-level latency tracking to promote low-latency computations. Jost et al. [8] augment a type system with cost values to extract upper bounds on the worst-case execution time and heap space usage. Their approach, however, targets embedded systems where both time and space bounds are important. Delange and Feiler [5] propose an incremental, model-based approach to analyze the validity of latency requirements in cyber-physical systems. Cohen et al. [4] present a type system raising the developer's awareness for inefficient code in terms of energie consumption. Their approach augments types by energy consumption patterns and uses type inference to track a program's energy consumption. Session types (e.g., Hu et al. [7]) have been successfully applied to distributed programming to check distributed protocols, but focus on protocol correctness rather than communication cost.

REFERENCES

[1] Andreas Abel. 2007. Type-based termination: a polymorphic lambda-calculus with sized higher-order types.

[2] Henk Barendregt, J Barwise, D Kaplan, HJ Keisler, P Suppes, and AS Troelstra. 1984. Studies in Logic and the Foundations of Mathematics. (1984).

[3] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-distribution of Actor-based Services. In *Proceedings of PACMPL (OOPSLA '17)*. ACM, New York, NY, USA.

[4] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. 2012. Energy Types. In *Proceedings of OOPSLA '12*. ACM, New York, NY, USA.

[5] Julien Delange and Peter H. Feiler. 2014. Incremental latency analysis of heterogeneous cyber-physical systems. In *Proceedings of REACTION '14*.

[6] Google Data Center FAQ. 2012. http://www.datacenterknowledge.com/archives/2012/05/15/google-data-center-faq/. (2012). Accessed 2018-04-16.

[7] Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP '08)*. Springer-Verlag, Berlin, Heidelberg, 516–541. https://doi.org/10.1007/978-3-540-70592-5_22

[8] Steffen Jost, Hans-Wolfgang Loidl, Norman Scaife, Kevin Hammond, Greg Michaelson, and Martin Hofmann. 2009. Worst-case execution time analysis through types. In *Proceedings of ECRTS '09*.

[9] Ulrich Kohlenbach. 2008. Applied Proof Theory - Proof Interpretations and their Use in Mathematics. In *Springer Monographs in Mathematics*.

[10] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. 2018. TCEP: Adapting to Dynamic User Environments by Enabling Transitions Between Operator Placement Mechanisms. In *Proceedings of DEBS '18*. ACM, New York, NY, USA.

[11] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: the Cost of Consistency. *IEEE Transactions on Software Engineering* (2018).

[12] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[13] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. 2017. Quality-aware Runtime Adaptation in Complex Event Processing. In *Proceedings of SEAMS '17*. IEEE Press, Piscataway, NJ, USA.

[14] Pascal Weisenburger, Tobias Reinhard, and Guido Salvaneschi. 2018. Static Latency Tracking with Placement Types FTfJP. In *Proceedings of the 20th Workshop on Formal Techniques for Java-like Programs (FTfJP'20)*.

[15] Mike P. Wittie, Veljko Pejovic, Lara Deek, Kevin C. Almeroth, and Ben Y. Zhao. 2010. Exploiting Locality of Interest in Online Social Networks. In *Proceedings of CoNEXT '10*. ACM, New York, NY, USA.