Exploring Algebraic Placement in Multiparty Languages

GEORGE ZAKHOUR, PASCAL WEISENBURGER, and GUIDO SALVANESCHI, University of St. Gallen, Switzerland

EXTENDED TALK ABSTRACT

This talk provides an overview of our ongoing research on the relationship between type systems and placement systems in programming languages for distributed systems.

In distributed systems, the placement of data and computation plays a crucial role in ensuring protocol correctness, fault tolerance, security of information flow, and performance optimization. Thus, researchers have explored various techniques to express and manage data placement. Type-based approaches have proven particularly effective in modeling places and their interactions. For example, choreographic programming [1, 2, 6] ensures safe communication protocols across different locations by modeling these locations – so-called roles – as types. Similarly, multiparty session types [4, 5] specify a communication protocol for message exchange over communication channels. Recent languages for multitier programming – a programming paradigm that provides language abstractions to specify the placement of data and computations on the different components of the distributed system – also opted for expressing placement in the type system [7, 9, 11].

Reification of placements into language-level concepts enables programmers to reason about which components perform computations and about the communication between them.

Challenges. While the different approaches for programming multiparty systems share the common objective of static reasoning across distributed parties, integrating such reasoning techniques into general-purpose programming languages remains challenging. Traditionally, placement features are retrofitted onto existing type systems. Interweaving the placement system with a traditional type system, however, leads to increased complexity in reasoning about placement correctness because the metatheory needs to refer to both systems at the same time. Thus, the formal model for the placement system needs to deal with semantics details that are fundamentally orthogonal to the placement system, impeding modular reasoning. Such interdependencies complicate the design of the placement system. More advanced features, like polymorphic [3, 8, 10] or dynamic placement [12], have been recognized as crucial functionalities in the domain, but their integration into programming languages lacks a principled formulation of the placement system that is independent of the type system and is dedicated exclusively to reasoning about placements.

Insight. In this talk, we argue that static semantics of the language is orthogonal to the idea of placements by demonstrating how a placement system (for checking placements) can be separated completely from a type system (for conventional type-checking) and how both can be composed and reasoned about modularly.

Our key insight is that the structure of a placement system is identical to the structure of a type system, enabling ideas from one to be applied to the other. Practically, this correspondence allows us to repurpose Haskell's type checker to become a placement checker. Theoretically, we use this correspondence to build a *placement algebra* on places akin to algebraic data types in conventional type systems.

Structure of the Talk. First, we introduce a Haskell embedding of a placement system that is independent of the type system and whose purpose is to reasoning about placements only. The key idea is to implement the placement checker and the type checker independently, which is achieved

Authors' address: George Zakhour, george.zakhour@unisg.ch; Pascal Weisenburger, pascal.weisenburger@unisg.ch; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, Torstrasse 25, St. Gallen, SG, 9000, Switzerland.

Table 1. Correspondence Between Type Systems and Placement Systems.

Type System		Placement System
Types Int, Boolean, (String → Char)	⟷	**Places** [2, 7, 9] Database, Cache
Parametric Polymorphism Set a, Map k a, Either a b	⟷	**Polymorphic Placement** [3, 10] LeaderElection p, DistList p0 p1
Sum Types Peano Numbers, Weekdays, Maybe	⟷	**Dynamic Placement** [12] Sum Cache DB, Geo-distribution
Product Types Tuples (a, b), Records { f0: a, f1: b }, Streams	⟷	**Data Sharing or Replication** [novel in this work] Conflict-free Replicated Datatypes, Configurations

by introducing types of the form Type @ Place. The type checker only inspects the left side of the @, leveraging the standard Haskell type-checking through the Glasgow Haskell Compiler (GHC). The placement checker, on the other hand, only inspects the right side of the @. Building on our insight that the structures of the placement system matches the structure of the type system, our Haskell embedding repurposes the existing Haskell type checker to check placements (but not "conventional" data types as the type checker does).

The following example shows a name value that is placed on a client using the on construct – where the value's placement is reflected statically by the @ ascription – and its remote access on the server using move (which may be read as "move here"):

```
name :: String @ Client
name = on Client $ "Rumpelstiltskin"

program :: String @ Server
program = on Server $ "Client name: " ++ move name
```

Second, we show that previously proposed static placement mechanisms align nicely with well-known typing abstractions. This correspondence is depicted in Table 1, associating common type system techniques (left column) with their interpretation in a placement system (right column). For instance, parametric polymorphism corresponds to polymorphic code that is parametrized over its placement (Table 1, second row) and sum types correspond to dynamic placement (Table 1, third row). Thus, the placement system can be seen as an alternate "type system" that does not check data types but data places. Both checks can be executed independently of each other.

In our Haskell embedding, for example, dynamic placement can be expressed as sum. The following code snippet shows the signature of some data that can be retrieved either from a server or from a cache – depending on whether it has been cached already – and a client that uses a match construct to inspect the runtime placement of the value:

```
data :: Data @ Sum Server Cache

program :: Result @ Client
program = on Client $
  match data
    (left  -> {- ... -})
    (right -> {- ... -})
```

Third, we complete this type–place correspondence with a missing placement interpretation of product types, which serve as a means to specify data that is shared or replicated across places (Table 1, fourth row).

In our Haskell embedding, for example, replication can be expressed as product. The following code snippet shows the signature of some data that is replicated to two different data centers and a client that uses a projection construct to retrieve the value from the northern data center:

```
data :: Data @ Prod NorthernCenter SouthernCenter

program :: Result @ Client
program = on Client $
  let value = proj0 data in
  {- ... -}
```

We illustrate that placement sums and products form an algebra – precisely a commutative semiring – analogous to sums and products in conventional type systems. To demonstrate that the axioms of the commutative semiring hold, we reinterpret equality in the axioms as isomorphism and construct two functions whose composition is the identity *of places*.

We believe this type–place correspondence can inspire future research, re-interpreting type system techniques in the context of placement.

Example. We illustrate the modular reasoning and the placement features discussed earlier in a single example extending the traditional book buyer-seller protocol.

First we observe that a book seller can either be the book publisher or a bookstore. Next we adopt the setting from [8] in which two buyers share the budget of a book. And finally we assume the book will be gifted to some third party.

```
transaction :: Place p => Title @Sum Publisher Bookstore -> Budget @ Prod Buyer Buyer -> Maybe Book @ p
transaction = {- ... -}
```

From the type system's point-of-view, the `transaction` function simply takes a book title, a budget, and produces a book when the budget is that of the cost of the title.

From the placement system's point-of-view, the `transaction` function takes a resource that is either placed on a `Publisher` or on a `Bookstore`, then takes a resource shared on two `Buyer` places and produces a resource placed on some polymorphic place `p`. The fact that `p` is a place is enforced by the constraint `Place p`. The placement system will not reason about the *datatype* of the resources, instead it is only concerned with the placement and the movement of these resources.

REFERENCES

[1] Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing – ICTAC 2022*, Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu (Eds.). Springer International Publishing, Cham, 212–237.

[2] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (jan 2024), 59 pages. https://doi.org/10.1145/3632398

[3] Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process polymorphism in choreographies. *Journal of Functional Programming* 34 (2024). https://doi.org/10.1017/S0956796823000114

[4] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, CA, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

[5] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (apr 2016), 36 pages. https://doi.org/10.1145/2873052

[6] Fabrizio Montesi. 2014. Kickstarting Choreographic Programming. In *Proceedings of the 13th International Workshop on Web Services and Formal Methods* (Eindhoven, Netherlands) *(WS-FM '14)*, Thomas Hildebrandt, António Ravara,

Jan Martijn van der Werf, and Matthias Weidlich (Eds.). Springer-Verlag, Berlin, Heidelberg, 3–10. https://doi.org/10.1007/978-3-319-33612-1_1

[7] Tom Murphy, Karl Crary, and Robert Harper. 2007. Type-safe distributed programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing* (Sophia-Antipolis, France) *(TGC'07)*. Springer-Verlag, Berlin, Heidelberg, 108–123.

[8] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP, Article 207 (aug 2023), 25 pages. https://doi.org/10.1145/3607849

[9] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[10] Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)* (London, UK) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 3, 29 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3

[11] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *Comput. Surveys* 53, 4, Article 81 (Sept. 2020), 35 pages. https://doi.org/10.1145/3397495

[12] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Safe Dynamic Placement with First-Class Placed Values. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 297 (Oct. 2023), 29 pages. https://doi.org/10.1145/3622873