Local-First Distributed Configuration (Experience Report)

Michael Sperber
sperber@deinprogramm.de
Active Group GmbH
Tübingen, Germany

Abstract

We have been developing a distributed application called *Lokalisierung* using functional programming since 2015. Lokalisierung automatically configures mobiles devices in a large organization of car-inspection shops. It uses peer-to-peer synchronization to distribute configuration data among the devices, without the use of a central server. This paper describes the original design and evolution of the system. Specifically, we highlight how the event-based architecture supports synchronization and enables flexibility for changes after the original rollout such as undoing configuration changes, and database thinning. We also reflect on design and implementation mistakes we made during its 10-year development.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Software architectures**; **Functional languages**.

Keywords: functional programming, software architecture, education

ACM Reference Format:
Michael Sperber. 2026. Local-First Distributed Configuration (Experience Report). In *Proceedings of the 4th ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '26), August 24–29, 2026, Indianapolis, IN, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3830438.3830957

1 Introduction

In 2015, we (Active Group, a software consultancy in Germany that uses functional programming for all of our projects) were tasked with implementing an application for a large car-inspection-shop organization for configuring its new fleet of over 1000 mobile devices in several hundred shops.

Each car-inspection shop has multiple lanes. At the end of each lane sits a desk closet with a printer and inspection equipment. Previously, the closet had also contained a desktop PC permanently hooked up to this equipment. The organization had decided to replace the closet-bound PCs by mobile devices (called *MEG*s), each assigned to an employee rather than a closet. Employees move around frequently, both between the lanes of a single shop and between shops.

Each time a MEG operates in a different lane, it needs to (wirelessly) connect to the equipment at that lane, and the inspection software needs to be reconfigured. Previously, this had been a laborious manual process—typing in MAC addresses, opening the printer configuration dialog etc.

Lokalisierung was commissioned to address this problem, to reconfigure the MEGs as they moved within and between shops. To that end, each MEG needs to know the configuration of each lane, which also changes as new shops get launched and equipment gets replaced. The organization lacked the infrastructure to operate a central server. Getting the data live from such a server would have been unrealistic anyway—car-shop WiFi is not the most reliable, and the organization's network did not have enough capacity. Still, it was desirable that configuration changes become visible near-instantly on the other MEGs in the same shop.

We designed an *append-only* data layer that stores *facts*, as opposed to traditional state-based databases. This design enabled us to tackle the project, resulting in a successful implementation, release and now decade-long operation.

Overview. We start by describing the database design in Section 2. The synchronization algorithm follows in Section 3, and conflict resolution in Section 4. Section 5 describes the time-machine facility for reverting changes. Section 6 recounts recent development on reducing database size and synchronization time. We summarize some missteps in Section 7, and conclude with lessons learned in Section 8.

2 Append-Only Database

The lack of a central server and the limited availability of network bandwidth means that each MEG needs to store and access configuration data locally. Moreover, the MEGs need to synchronize. This exposes Lokalisierung to the *CAP theorem* [4], which says that a mutable database cannot provide consistency (every read receives the most recent write), availability, and partition tolerance (continued operation when nodes disconnect from the network) simultaneously.

Here is the problem: A lane is configured to be associated with printer A, which breaks. A user changes the configuration to point to printer B on MEG M_1. MEG M_2, still configured to printer A, synchronizes with M_1, but should

FUNARCH '26, Indianapolis, IN, USA

ACM ISBN 979-8-4007-2868-6/2026/08
https://doi.org/10.1145/3830438.3830957

it just replace printer A by B in its printer database? How does it know which printer is the correct one? There might be timestamps associated with the two different printer configurations, and we might pick the newest one. Alas, as we would find out, changes would often happen in close temporal proximity and independently of one another, leading to conflicts that could not be resolved through timestamps. Just storing the seemingly "current configuration" would lead to confusing and difficult-to-trace situations where somebody's configuration would be overriden without explanation.

Consequently, instead of a conventional state-based database for the configuration data, we chose an approach based on storing *facts*. While state changes, a fact is something that remains true and is thus immutable. To illustrate the difference, "lane 2 has the FooJet printer with IP 5.5.5.5" is state, whereas "Mike said on May 21, 13:44 that lane 2 has the FooJet printer with IP 5.5.5.5" is a fact. Thus, the database is an evergrowing set of facts, from which the current state is derived, thus modeling the interaction of state with time. This corresponds to the notion of *event sourcing* [3, 9], as our facts effectively document events. (The term "event sourcing" was not known to us at the time.)

The pool of facts may have several facts pertaining to a single configuration setting. We use causal order to disambiguate: If a fact F_2 is created that affects the same setting as a fact F_1 *that is in the local database when* F_2 *is created*, we note that F_2 *obsoletes* F_1.

While there are specialized event databases, we chose SQLite for storing the facts [8], which requires no client-server setup and has extensive tool support. Here is the initial database schema. Table `kv` stores the facts themselves, and table `hashes` stores the obsoletion relationship:

```
CREATE TABLE kv (
  hash      BLOB   PRIMARY KEY NOT NULL UNIQUE,
  guid      BLOB   NOT NULL,
  property  STRING NOT NULL,
  value     BLOB   NOT NULL,
  meta      STRING NOT NULL);
CREATE TABLE hashes (
  key_hash BLOB NOT NULL,
  obsoleted_hash BLOB NOT NULL,
  FOREIGN KEY(key_hash) REFERENCES kv(hash));
```

The `guid` field contains a unique id that identifies the subject of a configuration setting, typically a specific car shop or lane. The `property` identifies a particular setting, such as `printerconfiguration`. Each fact comes with metadata in JSON format noting user, machine, and time. The `hashes` table relates "newer" facts to the ones they obsolete. A view combines both tables into a cleaned-up table of current facts:

```
CREATE VIEW kvCurrent AS
SELECT hash,guid,property,value,meta FROM kv
WHERE kv.hash NOT IN
  (SELECT obsoleted_hash FROM hashes)
```

With this schema in place, we built a database-access layer in the code itself, written in F#. While it would have been possible to just use SQL directly from the application code, this has several architectural drawbacks:

- The SQL is coupled to the database schema, making it difficult to change.
- A direct reference to a SQLite library would couple the application code to SQLite.

While .NET's ADO.NET framework could have loosened this coupling, it still would have required *some* ADO.NET-supported database. We decided early on that we wanted to be able to use the database access layer without SQLite, using simple F# collections, particularly for testing. (This would become more relevant later in the project, see Section 5.)

Thus, we needed a mechanism for *dependency injection* [2]. We used a free-monad approach [10] with a data type for database operations:

```
type 'a Op =
 | Return of 'a
 | Get of GuidT * PropertyT * (ValueT [] -> 'a Op)
 | Put of FactT * (HashT -> 'a Op)
 ...
 | Atomically of ('a Op) Op
```

The `Atomically` operation is a scoped higher-order operation [11] for transactions. The `bind` function is standard, as is the builder for F# *computation expressions* [7]:

```
let atomically (op: 'a Op): 'a Op =
  Atomically (bind op (fun v -> Return (Return v)))
let rec bind<'b, 'a> (opB: 'b Op) (f: 'b -> 'a Op)
      : ('a Op) =
    match opB with
    | Return b -> f b
    | Get (g, p, cont) ->
      Get (g, p, (fun v -> bind (cont v) f))
    | Put (d, cont) ->
      Put (d, (fun v -> bind (cont v) f))
    ...
    | Atomically op ->
      Atomically (bind op (fun op' ->
                           Return (bind op' f)))
type DbBuilder() =
    member _this.Return(v) = Return v
    member _this.ReturnFrom(m) = m
    member _this.Bind(m,f) = bind m f
let db = DbBuilder()
```

Computation expressions allow writing the application code in a straightforward, sequential way, while clearly limiting database access code to functions with `Op` return type. (We did not make use of the features of computation expressions beyond monadic notation.) The codebase includes two different interpreters for `Op` values, one accessing SQLite, one using in-memory F# collections.

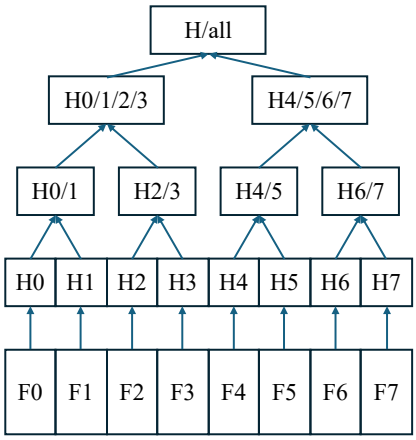

Figure 1. Merkle tree

3 Synchronization

With the facts database and obsoletion relation in place, it becomes feasible to synchronize the facts between two MEGs efficiently. The standard method for this is a *Merkle tree* [6]:

A Merkle tree sits atop the set of facts, each of which has a hash—see Figure 1. The individual hashes H_i of facts F_i form the leaves of the tree. Going up the tree, each node is labelled with a hash that is the result of combining the hashes from its child nodes. This means that the hash that labels each node summarizes the facts below. The top hash summarizes the entire set of facts.

The synchronization algorithm implicitly constructs the Merkle tree for both MEGs involved, using the bits of the fact hashes as paths down the tree. The algorithm traverses the tree top-down, sending hashes to the other side, and keeping a set of candidate hashes whose facts might need transmitting to the other side.

If the top hashes agree, no synchronization is necessary. Otherwise, both sides descend a level, and each sends its hashes to the other side as potential candidates. The hashes that agree can be removed from the candidates, and the algorithm descends to the next level only from the candidates that remain. When the algorithm finally reaches the lowest level, the remaining candidates identify those hashes that need to be transmitted to the other side.

An MEG triggers synchronization by sending UDP broadcasts containing the top hash into the local network

- when the MEG enters a network (typically when the car containing it drives onto the parking lot of the shop),
- when the network configuration changes,
- after new facts were added.

When an MEG receives such a broadcast, it compares the broadcast's top hash with its own, and, if they differ, engages in a (TCP-based) synchronization with the other side. The number of concurrent synchronizations is bounded by the number of MEGs in a shop, typically less than 20, which both network and MEGs can easily handle. This tactic has proved exceedingly effective in keeping all MEGs sufficiently updated to be usable wherever they go.

The synchronization algorithm is organized into three components:

Algorithm This implements the synchronization algorithm as a pure function.

Protocol This implements serialization of the messages required by the algorithm.

Communication This orchestrates the two to implement synchronization between MEGs.

The only dependency between Protocol and Algorithm is the common message datatype, making both independently testable:

```
type Msg =
| IHaveFingerprints of MerkleFingerprintSet
| HereAreBlockHashes of list<Hash.T>
```

The `IHaveFingerprints` message tells the other side for each successive level the *fingerprints* of the candidates. A fingerprint combines the hash of the corresponding node with the path leading to that node from the root. When the algorithm reaches the lowest level, it generates a `HereAreBlockHashes` message containing the hashes of the blocks to be transmitted to the other side. The Protocol layer resolves these hashes and directly transmits the associated blocks. The signature of the algorithm function is as follows:

```
let synchronizationRound (allHashes: Set<Hash.T>)
                         (treeSet: MerkleTreeSet)
                         (msg: Msg)
 : Choice<Msg * MerkleTreeSet, list<Hash.T>> = ...
```

The function takes as arguments the set of candidate subtrees and the message that just came from the other side, and returns either a message to transmit to the other side along with a set of candidates for the next round, or the blocks received from the other side.

The implementation of `synchronizationRound` is fairly involved, and many mistakes were made implementing it. However, the principal correctness criterion of synchronization is a single QuickCheck property [1] shown in Figure 2. The `checkSynchronizeWorked` function takes four sets of blocks, `blocks1` and `blocks1'` being the blocks present and received on one side, and `blocks2` and `blocks2'` on the other side. This has been the main test for validating the implementation before its original deployment in 2015, and no bugs have been detected since.

Note that this test is so straightforward because the synchronization algorithm is de-coupled from the rest of the synchronization, and because the algorithm itself presents a purely functional interface.

For the synchronization protocol of the first version, we used the FsPickler library [5], which provides a convenient

```
let checkSynchronizeWorked (blocks1: list<Block<'A>>) (blocks2: list<Block<'A>>)
                           (blocks1': list<Block<'A>>) (blocks2': list<Block<'A>>) =
  let hashes1 = List.map blockHash blocks1 |> Set.ofList
  let hashes2 = List.map blockHash blocks2 |> Set.ofList
  let hashes1' = List.map blockHash blocks1' |> Set.ofList
  let hashes2' = List.map blockHash blocks2' |> Set.ofList
  let allHashes = Set.union hashes1 hashes2
  Set.union hashes1 hashes1' |> should equal allHashes
  Set.union hashes2 hashes2' |> should equal allHashes
  Set.isSubset (Set.ofList blocks1') (Set.ofList blocks2) |> should be True
  Set.isSubset (Set.ofList blocks2') (Set.ofList blocks1) |> should be True


[<Test>]
member x.``synchronization works for arbitrary lists of blocks`` () =
    Prop.forAll(Arb.from<Set<Block<byte[]>> * Set<Block<byte[]>>>) (fun (bs1, bs2) ->
        let (blocks1, blocks2) = (Set.toList bs1, Set.toList bs2)
        let (blocks1', blocks2') = synchronize schema1 blocks1 blocks2
        checkSynchronizeWorked blocks1 blocks2 blocks1' blocks2') |> AssertProperty
```

Figure 2. QuickCheck test for synchronization algorithm

set of combinators for serializing and de-serializing F# values. We foresaw future changes such as incompatible changes to the FsPickler protocol, and prefixed synchronization messages with a header that did not go through FsPickler, containing a version number. To improve stability of the protocol, we implemented our own serialization library, which also turned out to be faster than FsPickler. The handshake at the beginning of synchronization would negotiate whether to use the new protocol—or the old one, if the Lokalisierung on the other side was still from the previous release. This turned out to be crucial when switching the serialization library, and when implementing a change to the hash function, documented in Section 7.

4 Conflict Resolution

If two users create different settings on their MEGs and these two MEGs synchronize, it is unclear to Lokalisierung which setting to use. Internally, Lokalisierung represents such settings as follows:

```
type Val<'a> when 'a : comparison =
| Absent
| Good of 'a
| Conflict of Set<WithMeta<'a>>
```

The `WithMeta` type attaches the metadata from the append-only database, allowing the dialog to display information about who made a conflicting setting, when and and where.

We ultimately settled on the following conflict strategy:

- If there are multiple facts for the same setting, Lokalisierung compares the values: If they are the same, it assumes that to be the correct setting. This eliminates most conflicts.
- When the values are different, the UI displays the relevant settings in red, but does not pop up disruptive dialogs. The user can then make a new setting, resulting in a fact obsoleting the conflicting facts.

This strategy proved acceptable to the users.

5 Data Modeling and Time Machine

Initial versions of Lokalisierung would retrieve settings individually from the append-only database, effect the corresponding configuration changes directly and transfer them to corresponding fields in the UI. The resulting coupling of the database structure to the UI structure, while an architectural problem in many applications, never proved problematic in Lokalisierung, mainly as the configuration structure has remained largely the same.

However, after Lokalisierung had been in use for a while, shop managers informed us that they often wanted to roll back erroneous settings. Fortunately, the append-only database contained the entire history of configuration changes, so all old settings were still present. Consequently, we set out to build a *time machine* that would allow users to go back to a past state for a given shop.

Note that this is not simply a matter of removing newer facts from the database: Synchronization would add them right back. Thus, Lokalisierung needs to make changes on top of whatever the current configuration is to re-create a past state. Then we could compare it to current state, and re-play the differences on top of the current state.

In order to re-create the past state, we needed a view on a subset of the facts. We did this by selecting the facts of the shop in question up to a certain point in time, and using that

to populate the in-memory implementation of the append-only database. (Which we had originally created only for testing purposes.) This required no changes to the codebase, and the dependency mechanism we had originally created for testing purposes paid off here as well.

In order to coherently represent and display the changes that would be made, we first created an explicit data model for a shop's configuration, as well as one for changes to it:

```
type Locality = {
  guid: PlaceGuid
  displayName: string
  macAddress: Val<option<MacAddress>>
  networkPrinterConfiguration:
    Val<option<PrinterConfiguration>>
  ...
}
type LocalityChangeDescription =
 | SetDisplayName of string
 | SetPlace of (PlaceGuid * PlaceChangeDescription)
  ...
type PlaceChangeDescription =
 | SetDisplayName of string
 | SetRank of int
 ...
```

These datatypes had previously been implicit, and Lokalisierung had turned configuration changes directly into facts. Users had to follow up mistakes with more changes. Making both the `Locality` representation and the changes explicit not only helped us implement the time machine. It also allowed us to decouple making the changes from persisting them. We made the regular configuration-change UI operate on the `Locality` datatype, create `ChangeDescription` values and display the corresponding changes, and offer an explicit "Save" button to persist them.

6 Database Thinning and Protocol Evolution

In 2024, the client asked us to do something about the size of the database, which had grown to several hundred megabytes. This was not so much a space problem as it made synchronization for new MEGs take a long time. Many of the facts being synchronized had long been obsoleted, and were not realistic candidates for restoration by the time machine. Thus, our client commissioned us to reduce the size of the database by deleting such irrelevant facts—*thinning*.

Deleting facts means deleting them from the `kv` table described in Section 2. As this table contained the `hash` primary key referenced by the `hashes` table, a migration of the database schema was necessary. The migration introduces a new table `all_hashes` containing just the hashes of all facts, with the new `kv` and `hashes` table referencing it:

```
CREATE TABLE all_hashes (
  hash BLOB PRIMARY KEY NOT NULL UNIQUE);
CREATE TABLE kv (
  ...
  FOREIGN KEY(hash) REFERENCES all_hashes(hash));
CREATE TABLE hashes (
  ...
  FOREIGN KEY(key_hash)
    REFERENCES all_hashes(hash));
```

With this new schema, Lokalisierung could remove facts from the database by just deleting them from `kv`. We settled on deleting facts that had been long obsoleted—beyond a certain age. The date of a fact's creating were recorded in the `meta` field of the `kv` table as a field in a JSON object, which SQLite fortunately can access directly. Consequently, thinning can be done with a single, quite fast, SQL `DELETE`:

```
DELETE FROM kv WHERE kv.hash IN
(SELECT kv_obsoleted.hash
FROM kv as kv_obsoleted, kv as kv_current, hashes
WHERE hashes.obsoleted_hash=kv_obsoleted.hash AND
      hashes.key_hash=kv_current.hash AND
     json_extract(kv_current.meta, '$.datetime')
       < :obsoletedBefore)
```

The `:obsoletedBefore` placeholder gets bound to the cutoff for the date of obsoletion.

7 Missteps

This section documents some architecturally relevant missteps that happened during development.

Overeager conflict resolution. During initial development, we assumed users would change settings mostly when in physical proximity to the lanes affected, and immediately propagate them throughout the car shop, making conflicts rare. Moreover, we assumed conflicts could be resolved on the spot. Lokalisierung would detect conflicts after synchronization and display a dialog requiring users to resolve them.

This strategy seemed obvious to us at the time, but proved unworkable in practice, as our assumption about the frequency and nature of conflicts proved false. From the users' perspective, Lokalisierung would pop up a dialog unpredictably, and on both MEGs involved in the synchronization, with no clear responsibility as to who should resolve it.

Our knee-jerk reaction was to automate filling out the dialog. This proved disastrous, as it would happen on both MEGs involved in the synchronization, creating two distinct resolution facts. These would be exchanged on the next synchronization, yielding another conflict, and so on, resulting in an avalanche of facts. Fortunately, this was detected within a few minutes of deployment and stopped.

Unstable hashes. The `obsoleted_hash` column in the `hashes` table has no `FOREIGN KEY` constraint. Indeed, for historical reasons there are entries with obsoleted hash codes with no counterpart in `all_hashes`. These entries contribute to the hash code of a fact, so we could not simply delete those

entries. This problem was compounded by another error made when we implemented the thinning procedure—we would use this SQL to insert new entries into `hashes`:

```
INSERT INTO hashes(key_hash, obsoleted_hash)
```

If the entry in question already exists, this creates a duplicate entry, which in turn would impact the hash of the corresponding fact, making the hashes in `kv` inconsistent, eventually leading to duplicate entries in `kv` that differed only in their hashes. Unfortunately, this problem was not spotted in testing, and shipping the corresponding release produced inconsistent databases that—addding insult to injury—quickly made the database grow *larger* than its original size.

In addition to fixing this bug, we also had to remove the duplicates from the database. This had to be done in a consistent fashion so that synchronization would not re-recreate the problem. Worse, the hashes are computed from the data in `kv`, and we had introduced the problem along with code that deletes entries in `kv`, making it impossible to re-compute the hashes of these entries.

We ended up changing the hash calculation to only depend on the data in `kv`, not on `hashes`. This avoids duplicate facts, but allows differences in obsoletion to persist between MEGs. To compensate for these, Lokalisierung treats facts introduced before this change differently from those after, only considering the "newest old" fact.

Incorrect projection. Lokalisierung identifies the shop an MEG is in by looking up its IP subnet. To that end, the first version of Lokalisierung contained a *locality cache*, a table mapping subnets to the GUID of its shop. (A *projection* in event-sourcing terminology.) Unfortunately, the code creating the cache contained a bug (never found) that would result in incorrect entries. The solution turned out to be to iterate over the facts with property `subnet`—the delay is not noticeable to users. We subsequently deleted the cache and the code maintaining it.

8 Lessons Learned

Functional programming and F# have served us well to keep the architecture open for extensions and changes over the years. Here are key lessons from the Lokalisierung project:

- The purely functional implementation of the core of the synchronization algorithm made it amenable to property-based testing. This enabled us to establish the critical correctness property of the algorithm.
- The event-based architecture enabled us to establish key functionality: Peer-to-peer synchronization and the time machine. Both would have been much more difficult with a traditional state-based database design.
- While conflict resolution is important, overeager resolution is bad.
- In a distributed system, conflicts occur even when the context of the system suggests that they should not.
- While events are a good basis for persistence, it is still a good idea to have an explicit data model in the code to implement more complex functionality such as the diff-based configuration UI, and the time machine.
- With event sourcing in place, it is not always necessary to implement projections in order to give efficient access to the current state. Some indexing of the facts/events may be sufficient.
- The database had uniqueness requirements that we had not encoded as constraints—we should have.
- Versioning the synchronization protocol from the beginning allowed us to evolve it over time.

Acknowledgments

I thank Andreas Bernauer for the early design of the add-only database, and Helmut Dobretzberger for maintaining Lokalisierung over the years.

References

[1] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, 268–279. doi:10.1145/351240.351266

[2] Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. https://martinfowler.com/articles/injection.html. Blog post.

[3] Martin Fowler. 2005. Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html. Blog post.

[4] A. Fox and E.A. Brewer. 1999. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. 174–178. doi:10.1109/HOTOS.1999.798396

[5] FsPickler [n. d.]. FsPickler: A fast, multi-format messaging serializer for .NET. https://mbraceproject.github.io/FsPickler/.

[6] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Carl Pomerance (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378. doi:10.1007/3-540-48184-2_32

[7] Tomas Petricek and Don Syme. 2014. The F# Computation Expression Zoo. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324* (San Diego, CA, USA) *(PADL 2014)*. Springer-Verlag, Berlin, Heidelberg, 33–48. doi:10.1007/978-3-319-04132-2_3

[8] SQLite [n. d.]. SQLite. https://sqlite.org/.

[9] Ben Stopford. 2018. *Designing Event-Driven Systems*. O'Reilly Media.

[10] Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758

[11] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2633357.2633358

Received 2026-06-01; accepted 2026-06-17; revised 12 July 2026