*JFP* **30**, e27, 50 pages, 2020.  
doi:10.1017/S0956796820000246

# *Typed dataspace actors*

SAM CALDWELL, TONY GARNOCK-JONES
and MATTHIAS FELLEISEN

*Northeastern University, Boston, MA 02115, USA*
(*e-mails:* samc@ccs.neu.edu, tonyg@leastfixedpoint.com, matthias@ccs.neu.edu)

**Abstract**

Actors collaborate via message exchanges to reach a common goal. Experience has shown, however, that pure message-based communication is limiting and forces developers to use design patterns. The recently introduced dataspace actor model borrows ideas from the tuple space realm. It offers a tightly controlled, shared storage facility for groups of actors. In this model, actors assert facts that they wish to share and interests in such assertions. The dataspace notifies interested parties of changes to the set of assertions that they are interested in. Although it is straightforward to add the dataspace model to untyped languages, adding a typed interface is both necessary and challenging. Without restrictions on exchanged data, a faulty actor may propagate erroneous data through a malformed assertion, causing an otherwise well-behaved actor to crash—violating the key principle of failure isolation. A properly designed type system can prevent this scenario and rule out other kinds of uncooperative actors. This paper presents the first structural type system for the dataspace model of actors; it does not address the question of behavioral types for assertion-oriented protocols.

## 1 Coordinating conversing actors

Languages in the tuple space family (Gelernter, 1985; Carriero *et al.*, 1994; Murphy *et al.*, 2006; Mostinckx *et al.*, 2007) address the need for manageable sharing between concurrent components (Tasharofi *et al.*, 2013).[1] Generally speaking, these languages connect concurrent processes to a storage medium, through which they communicate indirectly. Rather than sending a message along a channel or to a mailbox, a process deposits it in the central repository, from which another process may later retrieve it. This indirection provides decoupling from identity (e.g. address or channel) and across time, allowing a process to receive a message "sent" before its own creation.

Over the past few years, Garnock-Jones *et al.* (2014; 2016; 2017) have developed the dataspace model, a promising addition to the actor family based on tuple spaces. Roughly speaking, their dataspace actors utilize a storage medium for sharing knowledge among a group of participants (see Section 2 for a recap). A dataspace actor shares facts by asserting them in this common space. These assertions remain associated with the actor. A dataspace actor interested in facts states its interest in assertions about those facts. It is the dataspace's responsibility to inform an actor of the appearance and disappearance of facts in which it has stated interest. More precisely, the dataspace sends actors notifications of changes to

[1] The Blackboard idea (Newell & Simon, 1972; Englemore & Morgan, 1988) is a similar approach to the problem in an object-oriented setting.

the state of facts, who in response may add or withdraw assertions or spawn other actors. Finally, the dataspace withdraws an actor's assertions by force when the latter raises an exception.

We have implemented the dataspace model as extensions of ECMAScript (ECMA, 2015) and Racket (Flatt & PLT, 2010), and we have created several applications in these extended languages. This experience shows that dataspaces are particularly well-suited for situations where actors participate in several conversations at once and where many actors participate in the same conversation:

- Assertions lend themselves toward group communication. New participants can seamlessly join an ongoing conversation. Dataspace routing keeps each actor up-to-date with a minimum of boilerplate.
- Dataspaces decouple actors from each other, allowing multicast communication without the burden of maintaining a handle (address or channel) for each peer.
- The link between actor and assertion provides a uniform and powerful mechanism for resource management in the presence of partial failure, a notoriously difficult aspect of concurrent programming.

These features eliminate many design patterns necessitated by conventional coordination mechanisms, say, in Erlang/OTP (Armstrong, 2003). The need for design patterns in traditional models becomes especially prevalent when code must deal with potentially faulty actors. See Section 2.2 for more details.

Unfortunately, implementations of the tuple space model, including dataspaces, have largely been untyped or unityped—embedded in a typed host language in principle, but circumventing the type system by assigning the same type to all messages and requiring casts. This situation deprives programmers of the potential for type-based error prevention, documentation, optimization, design, and so on. The decoupling of components and general open-endedness of communication poses a challenge for type systems, which must predict the shape of incoming data. Our experience has also drawn our attention to fault isolation and scalability, two essential principles of actor programming, that suffer from the absence of a type discipline:

- Localizing faults to the originating component is an important tool for developing applications that detect and recover from exceptional situations. Typically, however, only some errors receive this treatment, leaving others open to transmission between actors via exchanged messages. Sadly, this can be as basic as a type mismatch in a message, such as a faulty actor that sends a two-element tuple where the recipient expects a two-element list. Only when the receiver tries to use that tuple as a list is the mistake detected, but now the exception occurs inside an actor that works as intended.
- Actor programming calls for concurrency—spawning an individual actor for each independent task. This philosophy implies certain architectural decisions for language implementations. In particular, the potential for a great number of simultaneously live actors means that they must share the underlying resources for computation, such as CPUs and threads. And where there is sharing, there is potential for abuse. Actors that take an overly long time to respond to messages can deprive peers of their own share of time. In the worst case, the event handler of

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

a running actor never terminates, permanently hampering the system and potentially rendering it unresponsive. Languages with type-based termination guarantees are a first step to face this challenge.

However, providing a typed interface to dataspaces proves a challenge. Type systems, by rule, control the exchange of data between components. By necessity, type system design is intimately connected to the underlying semantics of data exchange, such as function call or method dispatch. When a new communication medium is introduced, such as a dataspace, it must be reconciled with the type system. Tuple space-style communication is particularly challenging because the decoupling of components and indirect communications make the flow of data less apparent.

In response, we develop a structural type system that accounts for communication based on dataspaces (see Section 3). Our approach should be generalizable to other languages in the tuple space family. With the type system in hand, programmers may enjoy the benefits of both dataspaces and a type discipline. The type system for the base actor language satisfies both a type soundness property and a termination guarantee (see Section 4). Using these theorems, we can finally verify the (type) soundness of a complete dataspace model (see Section 5). The properties of the model have a number of practical implications for programming, as well as limitations (see Section 6). A prototype implementation of our type system confirms its basic usefulness, both in terms of convenience of use and of programming support (see Section 7).

## 2 Programming in the dataspace model

The dataspace model of actors is parameterized over the base language, that is, the language for programming each individual actor. For an introduction using illustrative examples, it is still best to use some concrete syntax. Here we use Racket syntax, falling back on one of the two untyped prototype implementations. This section provides an overview of the basic concepts (Section 2.1), illustrates it with examples (Sections 2.2 and 2.3), and makes the case for introducing a structural type system (Section 2.4).

### *2.1 Racket dataspaces*

In the dataspace model, an individual actor combines a private state with a behavior function. The private state is any piece of data from the underlying language. In a chat server program, for example, a room-list may have the shape

```
(list "FBI" "CIA")
```

recording the name of each active chat room. Intuitively, an actor behaves as if it were a function that maps events and the actor's current state value to a new state value and some instructions to carry out on its behalf:

$$\mathit{Event} \times \mathit{State} \rightarrow \mathit{State} \times \mathit{Instructions}$$

Instructions are interpreted by the surrounding dataspace, which connects actors with the goal of enabling conversation and managing their state.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

A dataspace equips a group of actors with a means of aggregating and sharing items of information, called *assertions*. In our chat server dataspace, we create one actor for each connection to an end user. This *user-agent* actor is responsible for relaying chat messages and carrying out commands from the remote user. For example, a connected user under the alias `"Mo"` may issue a request to join the room named `"FBI"`. In response, the user's agent actor states the following assertion:

```
(in-room "Mo" "FBI")
```

In the syntax of our Racket prototype, this assertion is stated as a Racket `struct` form. Racket structures are fixed-length tuples declared by the programmer, where the name (`in-room`) serves as an identifying tag. When the user leaves the room, the user-agent withdraws this assertion. Assertions range over basic values—numbers, strings, and so on—as well as first-order, immutable data structures, including lists and program-specific structures. Assertions have the same status in dataspace programs as facts in a Prolog database (Clocksin & Mellish, 1981).

The dataspace model links an actor with its assertions in a particular way. An assertion is *read-only*, and only the actor making an assertion may remove it. While multiple actors may make identical assertions, dataspaces hide such redundancy by providing a set view of an underlying bag of assertions. Additionally, the lifetime of an actor bounds the lifetime of its assertions. When an actor terminates, the dataspace removes its current assertions.

In order to observe the appearance and disappearance of assertions, an actor makes an *assertion of interest*. An assertion expressing interest in the above chat-room assertion uses the built-in structure `observe` (Figure 2 summarizes the extensions to base Racket):

```
(observe (in-room "Mo" "FBI"))
```

This particular assertion of interest could originate from an actor managing the room list. The dataspace notifies this actor—and any other actor expressing interest—when the user-agent asserts `(in-room "Mo" "FBI")`, and again when it withdraws the assertion.

Actors often assert interest in entire families of related assertions. For example, the room-list actor tracks the presence of every user in the `"FBI"` room, not just `"Mo"`. To express interest in *all* such values, the actor uses the wildcard ⋆ to assert

```
(observe (in-room ⋆ "FBI"))
```

In response, the actor receives a notification each time a matching `in-room` assertion appears or disappears. Semantically, a wildcard creates infinite sets of assertions; routing of notifications can thus be understood in terms of set intersection.

Figure 1 provides a visual intuition for the chat server dataspace described thus far. The `"Mo"` user has recently joined the `"FBI"` room. Accordingly, the user-agent actor places an `in-room` assertion in the dataspace. Because the room-list actor's assertions include an expressed interest in `in-room` assertions, the dataspace routing mechanism duly recognizes the overlap and sends an event.

An expression of interest is itself an assertion, observable by other actors. A common use of this recursion concerns *demand matching*. An actor capable of matching demand for $c$ asserts `(observe (observe `$c$`))`, while actors interested in $c$ assert `(observe `$c$`)`. The dataspace routing mechanism duly informs the first actor of the matching interest.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

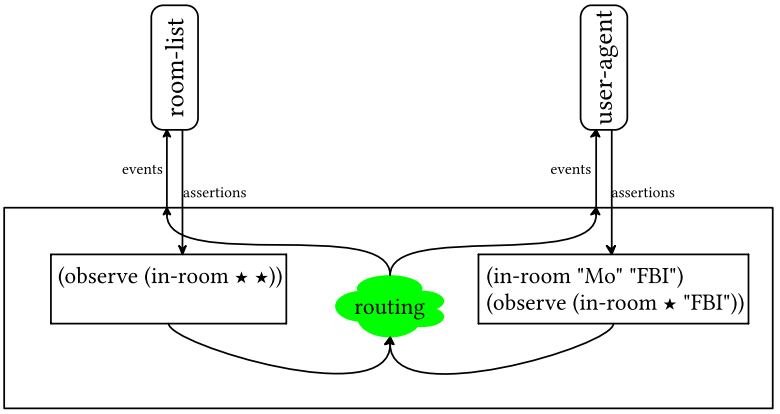


Fig. 1. Dataspace with actors engaged in a conversation.

In response, it performs the computation to produce the assertion *c*. Finally, dataspace routing again kicks in to notify the interested actor in the appearance of *c*.

Actors receive notification *Events* regarding changes to their dataspace's assertions in the form of a `patch`, which contains two sets. The first is a set of assertions newly *added* to the dataspace matching the actor's expressed assertions of interest. The second is a set of relevant assertions that have been *removed* from the dataspace. Hence, the room-list actor receives the notification

```
(patch {(in-room "Mo" "FBI")} ∅)
```

when user `"Mo"` joins. When the user leaves the room, the actor receives the corresponding notification `(patch ∅ {(in-room "Mo" "FBI")})`.[2]

Expressions between curly braces `{e ...}` create sets of assertions while actors analyze incoming sets using `project`, which resembles set comprehension. Projection takes the form `(project [`*pattern set*`]` *body*`)`, as in

```
(project [(in-room $name "FBI") {(in-room "Mo" "FBI")}] name)
```

The pattern serves to filter and destructure assertions in *set*. Evaluation instantiates the body of the `project` form once for each assertion in the set matching the given pattern, yielding a list. Here, it yields `(list "Mo")`. In addition to binding variables, prefixed with `$`, patterns may include the discard symbol `_` which matches any value. Since assertion sets may be conceptually infinite, some `project` expressions may yield infinite lists, as in `(project [$x {⋆}] x)`. The implementation signals an error when this happens.[3] The discard pattern `_` is especially useful for analyzing such sets; the expression `(project [_ {⋆}] "match")` yields the value `(list "match")`.

[2] Dataspaces additionally support *message broadcast*, though we focus here on communication through assertions. Messages may be thought of as a special case of assertion-based communication: sending a message atomically combines the assertion and immediate retraction of its value, i.e., `(message` *c*`)` behaves roughly as `(patch` *c* ∅`)` atomically followed by `(patch` ∅ *c*`)`.

[3] An alternative interpretation is that of a diverging computation, instantiating `x` an infinite number of times. Since these infinite sets are recognizable, we opt for signaling an error.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Actor behavior functions actually return values belonging to a disjoint sum, as opposed to the simplistic product described above. The sum allows an actor to shutdown in an orderly, non-exceptional manner. When an actor has fulfilled its purpose, it submits the first form of response, a `quit` record, to the dataspace:

(`quit` *instructions*)

In response, the dataspace removes the actor, withdrawing each of its assertions, and carries out some final *instructions* on its behalf. In the course of processing a notification, an actor may raise an exception. Uncaught exceptions are translated to `(quit empty)` by the dataspace sub-language.

The second form of return value makes or withdraws assertions via `patch`-es or augments the program with an entire new actor. In either case, the actor submits

(`transition` *state instructions*)

Such a `transition` record provides a new, updated private state for the actor and a list of *instructions* for the dataspace to carry out, in order. An instruction might be a simple patch. For example, the user represented by the `"Mo"` actor may issue a request to switch to the `"CIA"` chat room. To this end, the `"Mo"` actor would update both its `in-room` assertion, thus notifying other actors of the change, and its assertion of interest, allowing it to monitor other users present in the new room. It accomplishes this goal with this transition:

```
(transition
  "CIA"
  (list (patch {(in-room "Mo" "CIA") (observe (in-room ★ "CIA"))}
               {(in-room "Mo" "FBI") (observe (in-room ★ "FBI"))})))
```

The first item updates the actor's private state value to `"CIA"`, recording the user's current room. The transition provides a single instruction, a patch manipulating the actor's assertions. The first set in the patch places the assertion `(in-room "Mo" "CIA")` in the dataspace as well as an assertion of interest in other users in the `"CIA"` room. The second set in the patch withdraws the corresponding assertion of presence and interest for the previous `"FBI"` room.

An actor might also request the creation of a new actor. A corresponding instruction is an actor specification, which takes the form

(`actor` *behavior state assertions*)

providing a description of the actor's behavior function as well as its initial private state and assertions.

Finally, the form

(`dataspace` *actors*)

launches a dataspace program with the given initial actors. Figure 2 summarizes the extensions to Racket for dataspace programming.

### *2.2 Putting the pieces together*

Figure 3 shows a sketch of the user-agent actor implementation for a chat room, plus some of the context. The `main` function launches a dataspace with the described user-agent,

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

| | |
|---|---|
| `(dataspace` *expr*`)` | Launch a dataspace program with some initial actors |
| `(actor` *expr expr expr*`)` | An actor with a behavior plus initial state and assertions |
| `{` *expr ...* `}` | Create a set of assertions |
| `(observe` *expr*`)` | An assertion of interest |
| `⋆` | Describes all possible assertions |
| `(patch` *expr expr*`)` | Create a patch of two assertion sets |
| `(transition` *expr expr*`)` | Update private state plus some instructions to perform |
| `(quit` *expr*`)` | Termination with some final instructions |
| `(project [`*expr pattern*`]` *expr*`)` | Iterate over the matching assertions in a set |
| `$`*name* | Pattern that matches anything and binds it to *name* |
| `_` | Pattern that matches anything |

Fig. 2. Extensions to Racket for dataspace programming.

Untyped

```
(define (main)
  (dataspace
   (list
    (create-user-agent "Mo" "FBI")
    (create-room-list ...)
    ...)))

(define (create-user-agent name initial-room)
  (actor
   ;; behavior: notify the user as peers enter and
   ;; leave the current room.
   (lambda (event current-room)
     (define arrivals
       (announce (patch-added event) current-room " arrived."))
     (define departures
       (announce (patch-removed event) current-room " departed."))
     (transition current-room
                 (append arrivals departures)))
   initial-room
   {(in-room name initial-room)
    (observe (in-room ⋆ initial-room))}))

;; Create user-notification patches based on
;; presence assertions in an event.
(define (announce assertions current-room message)
  (project [(in-room $who current-room) assertions]
    (patch {(user-notification (string-append who message))} {})))
```

Fig. 3. A user-agent actor.

room-list, and any other potential actors. The actor's initial state establishes the presence of user `"Mo"` in room `"FBI"`, as described above. The behavior of the actor analyzes patches from the dataspace, projecting out the first slot from `in-room` assertions. The actor creates a `user-notification` for each arriving and departing peer. Another actor, not depicted, transforms such notifications into suitable network messages.

**Highlighting Dataspaces.** The chat example illustrates several benefits of dataspace coordination over pure message passing. Garnock-Jones (2017) surveys the comparative advantage of dataspaces versus other concurrency paradigms in greater depth, using different implementations of a similar chat example.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

- The *temporal duration of assertions*, i.e., the fact that an assertion remains in the dataspace until the originating actor withdraws it or terminates allows new actors to seamlessly join an ongoing conversation. When a new end user connects to the chat server, the program spawns a new user-agent actor on its behalf. This new actor expresses an interest in other members present in the current room. The interest effectively acts as a query of existing matching assertions. In response, the dataspace provides a description of all members present in the room.
- After joining a room and learning of existing members, the *same* assertion of interest provides the user-agent with ongoing *incremental knowledge updates*. The actor receives notification each time a current user leaves or a new user joins the room.
- *Group communication* simplifies the protocol and implementation of presence communication. The user-agent actor makes a single assertion that informs every peer to whom it is relevant. Assertions decouple actor identities. They allow communication without knowing exactly which actor(s), or even how many, to address. In traditional point-to-point models, a designated actor must maintain a collection of addresses or channels to enable communication among the other actors. As Joe Armstrong, designer of Erlang points out (Armstrong, 1994) (Chapter 8.7), "If we want to send a message to a process, then we need to know its PID. This is often inconvenient since the PID has to be sent to all processes in the system that want to communicate with this process." In the same book (Chapter 11), a similar multi-room chat example requires an additional actor for each possible room to keep track of this information.
- The *link between failure and communication* provides a uniform and convenient method for resource management in the presence of faulty actors. In this example, we may consider the ability to communicate with a particular chat user as a resource. As the availability of this resource is announced via assertion, it is automatically tied to the lifetime of the corresponding actor. An exception in the actor results in the withdrawal of the assertion, notifying interested peers. From the perspective of other user-agents, both orderly and exceptional termination convey the same information—that a certain peer is no longer communicable. Other actors in the system, meanwhile, can react in different ways, perhaps to clean up any associated network connection(s).

### *2.3 The role of recursive interest in dataspace protocols*

Dataspace protocols benefit from the use of recursive subscriptions, that is, assertions of interest in assertions of interest (and so on).

To illustrate the point, consider an "arithmetic service" dataspace. Assertions take the form `(sum x y z)` where $z = x + y$. Clients of the service make requests by asserting interest in sum assertions with particular numbers for $x$ and $y$, such as

```
(observe (sum 4 5 ⋆))
```

The actor implementing the service listens for requests with

```
(observe (observe (sum ⋆ ⋆ ⋆)))
```

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

in this case responding `(sum 4 5 9)`. The arithmetic service answers requests by analyzing the *x* and *y* values of sum requests and ignoring *z*:

```
(project [(observe (sum $x $y _)) e] (sum x y (+ x y)))
```

The example illustrates how interpreting the assertion of interest as a request yields the simplest possible protocol. Abstractly, this protocol for request/response situations combines the request assertion with the interest in the result. It thus reduces both the number of assertions an actor must state to make a request and the different types of assertions comprising the conversation. To appreciate the protocol's simplicity, consider the next-best alternative, which would be to use two distinct structs. One struct would represent requests separately via a `(sum-request` *x y*`)` assertion. The other would describe the results of requests as a `(sum` *x y z*`)` assertion. Assertions of interest in results must include the inputs *x* and *y*, so that there is no confusion with answers to other requests. This requirement is equivalent to `(observe (sum` *x y* `⋆))`, which is the request from above and suggests the simplification to a single struct.

### *2.4 The problems of programming actors in untyped languages*

Not surprisingly, using an untyped functional language as the implementation substrate of dataspace actors leads to the serious problems mentioned in the introduction.

(1) Productive communication between actors requires an agreement on the kinds of data to be exchanged; programming in such a manner is analogous to following a type discipline. Untyped languages like our Racket prototype provide little help with finding, and no help adhering to, a type structure that governs the communication between components. Consequently, the potential for mismatched messages undermines the principle of failure isolation. A buggy actor may make utterances that violate these expectations, but it is the actors that interpret the faulty messages that suffer the consequences. Worse, some of these mistakes correspond to simple type errors. A bug in a user-agent actor may turn a numeric user input into an assertion about its name, as a number rather than a string, causing other actors to crash while merrily continuing its own existence.

(2) The recursive nature of (interest in) assertions complicates the matter further. In the arithmetic dataspace from Section 2.3, a logic programmer may scan the description of sum assertions and surmise that the protocol lends itself to computing differences as well as sums, as with `(observe (sum 4 ⋆ 9))`. Such a request is incompatible with the implementation of the arithmetic service described above; it violates the assumption that the *y* slot is a number, even though it matches the service's assertion of interest `(observe (observe (sum ⋆ ⋆ ⋆)))`. As discussed in Section 2.1, `project` errors when a pattern variable has an infinite number of matches, so delivering such an assertion to the service actor causes it to crash—another failure to isolate misbehaving actors.

(3) The principle of scalability requires a certain amount of cooperation between actors in terms of execution time. Indeed, Garnock-Jones *et al*. (2016) show several key properties of the dataspace model under the assumption that the behavior function of every actor is total (including exceptions). However, programming actors in an untyped functional language such as Racket or ECMAScript allows the creation of both total and divergent behavior functions, with no way to distinguish the two. Moreover, divergence is a form

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

of failure, but it bypasses the reasoning mechanism provided by the language for these situations—automatic withdrawal of assertions. The termination guarantees of a simple type system aid in isolating and responding to failures as expected in the actor model.

(4) Additionally, developers may wish to place specific restrictions on the behavior of certain actors. For instance, in the case that some rooms are private, the developer of the chat server may wish to ensure that user-agent actors do not make overly broad queries. Permitting such queries would allow a curious user to discover the existence of, and join, private channels without an invitation. Types are one mechanism for imposing such constraints. If the type of user-agent actors indicates that no such subscriptions are made, the protocol is safe from intrusions.

## 3 Types for the dataspace model

Designing a type system for dataspaces requires consideration of the semantics of both communication and computation, with special attention to the intersection of the two concerns. Our design meets this high-level criterion as follows:

- The language of types accounts for the flexible nature of dataspace communication with "true" (set) unions (Pierce, 1991) for describing sets of assertions. Dataspaces are about commingling actors: each actor partakes in, and each assertion potentially pertains to, several conversations. Union types mirror the overlapping conversations, making them suitable for describing the (sets of) assertions in a dataspace.
- Communication and computation coincide in the functions used to express actor behavior. Dataspace event dispatch (communication) becomes function application (computation). The return values of behavior functions give rise to routing events. These functions warrant additional checking on inputs and outputs, accounting for their dual purpose.
- Additionally, behavior functions must terminate on all inputs. But, writing actors in a language with recursion permits diverging programs. To address this issue, we employ the standard method of basing the type system on the simply-typed $\lambda$-calculus plus an induction schema per recursive type (Jeuring, 1995), which is known to make for a terminating language.
- Finally, the computational constructs for creating and accessing the sets of assertions used for communication introduce new possibilities for computational errors. Hence, the type system prohibits creating assertion sets with higher-order data or selecting an infinite branch of an assertion set with `project`.

The rest of this section is an informal introduction to this somewhat unusual combination of ideas.

### *3.1 Typed dataspaces by example*

**Unions.** Union types are the basic building block of typed dataspace actors. In dataspaces, groups of actors participate in conversations where each utterance takes the form of

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

an assertion. Describing these conversations with types means grouping together related assertions. Unions capture the multi-party, overlapping nature of dataspace conversations.[4]

For instance, in the chat room example from Section 2, actors communicate presence information through `in-room` assertions and interest in those assertions. Each `in-room` assertion has type `(InRoom String String)`. A value of type `(InRoom String String)` is an assertion `(in-room` $v_1$ $v_2$`)` where both $v_1$ and $v_2$ are strings. By convention, type constructors such as `InRoom` are the camel-cased name of the described structure.

An assertion of interest in other participants, such as `(observe (in-room ⋆ "FBI"))`, has type `(Observe (InRoom ⋆ String))`. The type uses the parameterization of the `InRoom` constructor to allow wildcard interest in usernames but limit interest to named rooms. As a type, ⋆ stands for all possible assertions, including concrete strings, numbers, or interests, as well as any infinite set of assertions arising from {...} (set creation) expressions containing ⋆.

The union type `PresenceAssertions` describes the "Room Presence" conversation, including both the assertion made by a user-agent to signal its presence in a particular room and the assertion of interest used to monitor its peers:

```
(define-type PresenceAssertions
  (U (InRoom String String)
     (Observe (InRoom ⋆ String))))
```

Together, they describe the possible utterances in the conversation between user-agent actors concerning chat-room presence.

Grouping together the types of each assertion pertaining to a conversation in a union provides useful documentation and aids in separate, modular programming of actors. While a single actor may express only a subset of the assertions, the flexible nature of union types allows the composition of overlapping conversations. For example, the room-list actor monitors `in-room` assertions and publishes a list of results. The `InRoom` assertions, the room-list actor's interest in them,[5] the published `RoomList`, and the interest used by other actors to learn the list form an "Available Rooms" conversation:

```
(define-type RoomAssertions
  (U (InRoom String String)
     (Observe (InRoom ⋆ ⋆))
     (RoomList (List String))
     (Observe (RoomList ⋆))))
```

The "Room Presence" and "Available Rooms" conversations overlap, with each `in-room` assertion playing a role in both. In one conversation, an assertion signifies the existence of a particular user, while in the other it signifies the existence of a particular room.

As Figure 3 shows, a user-agent actor performs additional communication to notify the connected user of specific events. This conversation comprises `user-notification` structures, described by the type

```
(define-type NotificationAssertions
  (U (UserNotification String)
     (Observe (UserNotification ⋆))))
```

[4] Such types do not account for temporal or substructural constraints on exchanges; see Section 6.

[5] The type of interest in presence employed by the room list actor, `(Observe (InRoom ⋆ ⋆))`, reflects the potential for wildcard interest in all possible rooms, unlike that of the user agent.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

```
(define-type ChatDataspace
  (U PresenceAssertions
     NotificationAssertions
     NetworkAssertions
     ChatAssertions
     RoomAssertions))
```

Fig. 4. Chat server communication type.

The full implementation includes other conversations for network connectivity, sending chat messages, and changing rooms, each with a similar type: `NetworkAssertions`, `ChatAssertions`, `RoomAssertions`, and so on.

By describing the conversations in isolation, each actor can be implemented in terms of only those conversations in which it participates. In the case of the user-agent described here, we may elide `NetworkAssertions`, among others:

```
(define-type UserAgentAssertions
  (U PresenceAssertions
     RoomAssertions
     NotificationAssertions))
```

**Dataspaces.** Taking the union of all the conversations, as shown in Figure 4, yields a type that describes the entire dataspace. Supplying that type to a typed dataspace constructor,

```
(dataspace ChatDataspace
  (actor ...)
  ...)
```

demands that all assertions have type `ChatDataspace`. The annotation is referred to as the *communication type* of the dataspace. The communication type is an agreement among the actors, both limiting each individual's actions as well as enabling typed reasoning about the behavior of peers. In the chat dataspace, the type permits the user-agent actor's presence assertion `(in-room "Mo" "FBI")`, of type `(InRoom String String)`, and interest `(observe (in-room ⋆ "FBI"))`, with type `(Observe (InRoom ⋆ String))`, because of their inclusion in the `ChatDataspace` type via `PresenceAssertions`. By contrast, the `ChatDataspace` type prohibits actors that assert `(in-room "Marvin" 42)`, because the second slot is not a string, or overly broad queries such as `(observe ⋆)`, because `⋆` is not a subtype of any of the constructors that appear under `observe` in `ChatDataspace`.

While the communication type restricts the assertions an actor may make, it also enables reasoning about the shape of assertions that may match an expressed interest. The key assurance of the `ChatDataspace` type is that any assertion matching an expressed interest in `in-room` assertions is an `in-room` struct where both slots contain strings. That is, in any set of assertions $a$ sent to an actor, the set corresponding to the names of connected users, $\{v_1 \mid \texttt{(in-room } v_1 \ v_2\texttt{)} \in a\}$, is a finite set of strings, and similarly for the set of values in the second slot, the names of inhabited rooms. Consequently, the behavior functions of actors in the dataspace may use `project` to analyze the names of individual users and rooms without triggering an error.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Actors and Simple Behavior Functions.** Every individual actor operates within a dataspace, making and withdrawing assertions, processing assertions, and spawning further actors, each of which also resides in the dataspace. In typed programs, the dataspace contains only assertions belonging to a specific type—the communication type $\tau_c$. Hence, $\tau_c$ plays a central role in checking individual actor specifications.

The type for actors must then reflect the type $\tau_c$ of the dataspace in which it is going to run. A developer can use this type to rationalize the initial assertions and, most importantly, the code for the behavior function. Specifically, the developer must ensure that the behavior function (1) produces only actions—assertions as well as spawned actors—that the dataspace's type permits; (2) is prepared to deal with any event to which the dataspace may apply it; and (3) spawns only actors that recursively obey these constraints, too.

A translation of these insights calls for equipping an actor term

$$\texttt{(actor}\ \textit{behavior}\ \textit{state}\ \textit{assertions}\texttt{)}$$

with an interface type. This step resembles the addition of a parameter type to lambda terms during the design of a simple type system for languages based on the lambda calculus. Since an actor communicates within a dataspace of some communication type, the best way to signal this assumption is with an annotation $\tau_c$ that requests this match:

$$\texttt{(actor}\ \tau_c\ \textit{behavior}\ \textit{state}\ \textit{assertions}\texttt{)}.$$

In contrast to function application, actor application is *implicit*. An `actor` term is used by submitting it to the dataspace as an action. In dispatching events, the dataspace applies the actor's *behavior* function and interprets the resulting actions. These semantics—dispatching events and interpreting actions—are not represented in the surface syntax of the program, meaning that it is not possible to check the use of an actor independently from its use in a dataspace. We can express this insight by assigning the type

$$\texttt{(Actor}\ \tau_c\texttt{)}$$

to an actor term. By implication, it becomes straightforward to check a dataspace term. All initial actors must have the type `(Actor` $\tau_c$`)` if $\tau_c$ is the type of the dataspace.

Validating that an actor has a given type proceeds according to the three steps above:

1. The assertions of an actor come from two sources: either as part of its initial assertions or as an action produced by its behavior function. For the former, we check that the type of the initial assertion set is included by the assumed communication type $\tau_c$. For the latter, recall the informal signature of behavior functions from Section 2.1:

   $$\textit{Event} \times \textit{State} \rightarrow \textit{State} \times \textit{Instructions}$$

   For the moment, let us simplify things even further by considering the behavior function as taking in assertions of some type $\tau_{in}$ and outputting assertions of another type, $\tau_{out}$:

   $$\tau_{in} \rightarrow \tau_{out}$$

   Conceptually, $\tau_{in}$ and $\tau_{out}$ are the essence of the *Event* and *Instructions* types, respectively. Validating the assertions stated by the actor then entails checking that $\tau_c$ includes each type of assertion from $\tau_{out}$.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

2. Dataspaces compute and route events according to expressed interests Hence, the assertions constructed with `Observe` in $\tau_{out}$ define a bound on possible events. Type checking uses this bound together with the rest of the assertions in the dataspace, represented by $\tau_c$, to predict the types of events produced by routing when the program runs. To make this prediction, the type checker takes the intersection between assertions of interest in $\tau_{out}$ with $\tau_c$. The result is a type describing all potential events the behavior function may be applied to, which must be a subtype of $\tau_{in}$. Concretely, in the case of the user-agent actor $\tau_{out}$ is

```
(U (Observe (InRoom ⋆ String))
   (InRoom String String)
   (UserNotification String))
```

   According to the type, the actor might assert `(observe (in-room ⋆ "FBI"))`. It would then receive a notification containing all `in-room` assertions with the string `"FBI"` in the dataspace. Inspecting the `ChatDataspace` type of Figure 4, the type of potentially overlapping assertions is `(InRoom String String)`. Consequently, the user-agent actor's behavior function input type $\tau_{in}$ must accommodate such assertions.
3. Finally, an actor may spawn other actors into its dataspace. If every one of these actors has type `(Actor` $\tau_c$`)`, they all obey the communication discipline of the surrounding dataspace.

**Manipulating Assertion Sets.** The type system prevents the creation of illegal assertion sets by stratifying types into two levels. The first level, *flat types*, corresponds to the plain data suitable for sharing in the dataspace: `Int`, `String`, and so on, as well as type constructors such as lists, structs, and unions when applied to other flat types. The second level is everything else—values that cannot easily be compared for equality, such as functions, objects, and actors. Typing a set-creation form `{e ...}` then checks that each element `e` has a flat type, i.e., *not* a function, object, or actor.

The type `(AssertionSet` $\tau$`)` describes a set of assertions of type $\tau$ arising from an expression `{e ...}`. A patch `(patch` $e$ $e$`)` is essentially a pair of assertion sets, thus the type `(Patch` $\tau$ $\sigma$`)` records the type of assertions to add, $\tau$, and the type of assertions to withdraw, $\sigma$.

We use the abbreviation `(Event` $\tau$`)` to stand for `(Patch` $\tau$ $\tau$`)`, signifying that dataspaces notify actors of both the appearance and disappearance of assertions matching an interest. An actor may request the retraction of an assertion it is not presently making, which is a no-op. An actor may make use of this fact by issuing an overly broad retraction, alleviating some responsibility for tracking currently made assertions. For example, an actor may submit the patch `(patch {} {⋆})` to withdraw all of its current assertions. To account for this fact, we use the abbreviation

$$\texttt{(Action } \tau \ \sigma\texttt{)} \stackrel{df}{=} \texttt{(U (Patch } \tau \ \star\texttt{) (Actor } \sigma\texttt{))}$$

to describe actor actions. The type allows any set of assertions to be withdrawn, as well as the potential to require spawned actors to operate at the communication type $\sigma$.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Typed assertion-set projection differs from the untyped version in several ways. The first is a syntactic change to patterns. In order to facilitate type checking, pattern variables come with a type annotation, as in `$name:String`.[6] The type of this pattern variable is then `(Bind String)`, while the `_` pattern has type `Discard`. Type-checking additionally verifies that the expressions within patterns are well-typed.

Detecting erroneous uses of `project` involves considering paths to binding patterns and ⋆ in potentially matching types. Recall that `project` raises an error when a binding pattern, such as `$name:String`, has an infinite number of matches in the given set, as in the following expression:

```
(project [(in-room $name:String "FBI") {(in-room ⋆ ⋆)}] name)
```

Conceptually, performing the operation requires iterating over the infinite set

$$\{v \mid \texttt{(in-room } v \texttt{ "FBI")} \in \{\texttt{(in-room ⋆ ⋆)}\}\}$$

The cause of the error is not simply because the given set is conceptually infinite. Often, the structure of the pattern provides enough information to discriminate most elements of the set. In the expression

```
(project [(in-room $name:String "FBI") {(in-room "Mo" ⋆)}] name)
```

the matched set,

$$\{v \mid \texttt{(in-room } v \texttt{ "FBI")} \in \{\texttt{(in-room "Mo" ⋆)}\}\} = \{\texttt{"Mo"}\},$$

is finite, thus evaluation poses no issues. An error occurs exactly when a binding variable in the pattern corresponds to a wildcard ⋆ in the assertion set. The type system therefore tracks both uses—the latter by assigning ⋆ the type ⋆—and analyzes the type of the pattern against the type of the contents of the set. The potential for the pattern to match assertions is determined by computing the overlapping elements of the types. A type error arises only if there is a common path through both types that may potentially match, leading to `(Bind` $\tau$`)` in the pattern and ⋆ in the set.

**Termination.** In order for dataspace programs to make progress, individual actors must terminate—either normally or via an exception—in response to every event. This assumption is easily violated when programming with general-purpose constructs such as functions. The type system disallows recursive functions, but still allows for using recursive data structures such as lists via inductive schemas. Recursive data structures must be used via an inductive eliminator in the shape of a folding loop.

**Actor Subtyping.** Subtyping for actor actions aids modular development of actors and permits type-level constraints to be imposed on individual actors, rather than the entire dataspace. For example, the user-agent actor can be developed using a communication type that describes only those conversations in which it participates:

```
(actor UserAgentAssertions ...)
```

[6] Our prototype implementation can infer these annotations, but we include them in examples here to clearly separate typed and untyped code.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

yielding a term of type `(Actor UserAgentAssertions)`. Ultimately, however, user-agent actors operate in a dataspace with communication type `ChatDataspace`. Hence, we must check that the difference between the two communication types does not invalidate the reasoning by which type checking the user-agent actor first succeeded.

Checking the `actor` creation action computes an intersection between the assertions of interest made by the user-agent actor with `UserAgentAssertions` to determine the events it might receive. In a dataspace with a different type, such as `ChatDataspace`, the actor's interests might match different types of assertions, which may potentially be incompatible with its behavior function type. The question, then, is whether the actor's interests lead to "surprising" events when operating in a different type of dataspace.

Actor subtyping answers this question. The first item to check is that all of the user-agent's assertions are allowed in the greater chat dataspace, that is, `UserAgentAssertions` must be a subtype of `ChatDataspace`. Next, we must make sure that running in a `ChatDataspace` context does not yield surprising events, i.e., events not considered the first time we checked the user-agent. The `UserAgentAssertions` type permits one type of interest,

$$\tau_{obs} = \texttt{(Observe (InRoom} \star \star\texttt{))}$$

Intersecting $\tau_{obs}$ with `ChatDataspace` yields the possible type of events the actor receives when run in the dataspace, `(InRoom String String)`.

Since `(InRoom String String)` is in `UserAgentAssertions`, it is also in the intersection of $\tau_{obs}$ and `UserAgentAssertions`. Consequently, `(InRoom String String)` events have already been considered, and deemed safe, against the input type of the user-agent actor's behavior function during the checking of the corresponding `actor` form.

### *3.2 Typing the chat room*

Figure 5 displays the typed version of the code from Figure 3, highlighting the changes. The primary difference between the two figures is the addition of the `UserAgentAssertions` and `ChatDataspace` type abbreviations (Figure 4). Otherwise, there are minor changes to insert type annotations on function parameters, binding patterns, and actor-spawning expressions. The example also makes use of the abbreviation $\bot$ for the empty union.

The behavior function comes with a narrow type of input event,

```
(Event (InRoom String String))
```

containing only one of the many forms of assertions with which actors communicate in the dataspace. Because the actor states only one type of interest, the type system is able to verify that incoming events do indeed have such a refined type. The example illustrates how a highly expressive type system can easily validate complex confluences of communication and computation.

The typed chat dataspace above rules out simple mistakes such as using a number instead of a string for a username. These mistakes can result in non-local actor failure: faults in actors that consume, rather than produce, bad presence information. The `ChatDataspace`

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Typed

```
(define-type UserAgentAssertions
  ... defined on page 12 ...)

(define-type ChatDataspace
  ... defined in figure 4 ...)

(define (main)
  (dataspace ChatDataspace
   (list
    (create-user-agent "Mo" "FBI")
    (create-room-list ...)
    ...)))

(define (create-user-agent [name : String]
                           [initial-room : String]
                           -> (Actor UserAgentAssertions))
  (actor UserAgentAssertions
   ;; behavior: notify the user as peers enter and
   ;; leave the current room.
   (lambda ([event : (Event (InRoom String String))]
            [current-room : String])
     (define arrivals
       (announce (patch-added event) current-room " arrived."))
     (define departures
       (announce (patch-removed event) current-room " departed."))
     (transition current-room
                 (append arrivals departures)))
   initial-room
   {(in-room name initial-room)
    (observe (in-room ⋆ initial-room))}))

;; Create user-notification patches based on
;; presence assertions in an event.
(define (announce [assertions : (AssertionSet (InRoom String String))]
                  [current-room : String]
                  [message : String]
                  -> (List (Patch (UserNotification String) ⊥)))
    (project [(in-room $who:String current-room) assertions]
      (patch {(user-notification (string-append who message))} {})))
```

Fig. 5. The typed user agent actor.

communication type prevents such an actor from being introduced into the typed dataspace by specifying that room names are only ever finite sets of strings.

**Constraining the User-Agent.** The typed chat dataspace permits queries over both the presence of users in a specific room, `(Observe (InRoom ⋆ String))`, and over all rooms, `(Observe (InRoom ⋆ ⋆))`. The former are used by user-agent actors to monitor the presence of peers in a room, while the latter are used to aggregate information about

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

which rooms exist. Since `UserAgentAssertions` includes both `PresenceAssertions` and `RoomAssertions`, the type permits user-agent actors to express both interest in specific room names and wildcard interest in every room. As discussed in Section 2.4, the developer may wish to enforce that user-agents do not make overly broad queries. For example, private channel names may leak to an actor that asserts `(observe (in-room ⋆ ⋆))`. To address this, the developer may ascribe a refined communication type to user-agent actors:

```
(define-type RestrictedUserAgent
  (U (InRoom String String)
     (Observe (InRoom ⋆ String))
     NotificationAssertions))
```

This restrictive type enforces that queries are only over *specific* room names. Actor subtyping permits using the refined user-agent actor in the chat server dataspace.

### *3.3 Revisiting the arithmetic service*

Recall the arithmetic service dataspace (Section 2.3) and the error that arises from ⋆ assertions. Types prevent the scenario where `sum` assertions are abused to request difference calculations. The program may use the communication type

```
(define-type ArithmeticAssertions
  (U (Sum Int Int Int)
     (Observe (Sum Int Int ⋆))
     (Observe (Observe (Sum ⋆ ⋆ ⋆)))))
```

and this type rules out difference-calculation requests.

Such a difference request assertion, say `(observe (sum 4 ⋆ 9))`, would have type `(Observe (Sum Int ⋆ Int))`, which is *not* subsumed by `ArithmeticAssertions`. In particular, because ⋆ stands for *all* types of assertions, including strings and structures, it is incompatible with the occurrence of `Int` in the corresponding position in `ArithmeticAssertions`. An actor with such an output fails to type check in the `ArithmeticAssertions` dataspace, thus preventing the dynamic error explained in Section 2.4.

Alternatively, we could have chosen a communication type that is more permissive with regard to interest in `sum` assertions:

```
(define-type PermissiveInterests
  (U (Sum Int Int Int)
     (Observe (Sum ⋆ ⋆ ⋆))
     (Observe (Observe (Sum ⋆ ⋆ ⋆)))))
```

A dataspace of this type allows difference requests. However, now a type error arises for the actor implementing the arithmetic service. Recall that the service iterates over incoming assertion sets *e*,

```
(project [(observe (sum $x:Int $y:Int _)) e] (sum x y (+ x y)))
```

This projection occurs inside the body of a behavior function, which assumes some type for the incoming assertions contained by *e*. If that *e* has type `PermissiveInterests`, the

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

type system signals an error, since the path to `$x:Int` in the pattern leads to ⋆ in the patch set. By contrast, assuming the contents of the set *e* have type `ArithmeticAssertions` allows the projection to type check.

When we try to create an actor action with the behavior function for a dataspace with communication type `PermissiveInterests`, the type system detects a mismatch between the assumption and reality. Concretely, the actor makes an assertion of interest `(observe (observe (sum ⋆ ⋆ ⋆)))` to learn about sum requests, with type

```
(Observe (Observe (Sum ⋆ ⋆ ⋆)))
```

Potentially matching assertions are the intersection with `PermissiveInterests`,

```
(Observe (Sum ⋆ ⋆ ⋆))
```

which is *not* a subtype of `ArithmeticAssertions`. Finally, checking the actor action succeeds if the supplied communication type annotation is `ArithmeticAssertions`. Actor subtyping prevents instantiating such an actor in the `PermissiveInterests` dataspace through a similar failed type-checking attempt.

The arithmetic service actor typifies the interplay between projection, function, and actor typing. When writing a behavior function, the developer analyzes incoming assertion sets using `project`. In order for the function itself to type check, the set must have a type compatible with the supplied patterns. These constraints flow outward, to the function's parameter for incoming events, and they then become part of the domain of the function's type. When the function reaches an `actor` action, the type system finds the assumptions in the domain of the function type and compares them with the reality of the surrounding dataspace. Only when all of these elements agree are programs well-typed.

## 4 The semantics of actors

### *4.1 Specifying an actor*

Figure 6 introduces $\lambda_{ds}$, a model language for articulating untyped dataspace actors. It is representative of our ECMAScript and Racket prototypes, i.e., functional languages that extend the λ-calculus syntax with means to interface with dataspaces.

A complete program in $\lambda_{ds}$ is a description of a dataspace, `dataspace` $M$. The expression $M$ computes a list of `actors` to launch at the start of the program. Over the course of execution, additional actors may be dynamically spawned through `actor` actions as well as removed due to failure or termination. Section 5 describes how this initial description yields a running actor system.

Extensions to the base functional model compute values that, when sent to the surrounding dataspace, trigger certain actions: an assertion of interest (`observe` $M$); an assertion of fact ($m(\overrightarrow{M})$); the spawning of a new actor (`actor` $M_b\ M_s\ M_a$). The three parts of `actor` correspond to the behavior function ($M_b$), private state ($M_s$), and initial actions ($M_a$) of an actor. Additionally, the extensions also include expression forms to create sets of assertions ($\{\ \overrightarrow{SK}\ \}$) and compose patches ($M^+/M^-$).

Behavior functions in $\lambda_{ds}$ return a pair of a new state value and a list of actions, as opposed to the `quit` and `transition` records of our prototypes described in Section 2.1.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rcl}
I \in \textbf{Init} & = & \texttt{dataspace}\ M \\
M \in \textbf{Expr} & = & \lambda x.\, M \mid M\ M \mid x \mid p\ \overrightarrow{M} \mid b \\
& \mid & \texttt{error}_{\eta} \\
& \mid & (\overrightarrow{M}) \mid \texttt{cons}\ M\ M \\
& \mid & \texttt{observe}\ M \mid m(\overrightarrow{M}) \\
& \mid & \{\ \overrightarrow{\mathrm{SK}}\ \} \mid M^{+}/M^{-} \\
& \mid & \texttt{actor}\ M_b\ M_s\ M_a \\
& \mid & \texttt{project}\ M\ \texttt{with}\ \mathrm{PAT}\ \texttt{in}\ M \\
m \in \textbf{Msg} & = & \ldots \text{message constructors} \\
p \in \textbf{Prim} & = & \ldots \text{primitives} \\
b \in \textbf{BasicVal} & = & \ldots \text{base values}
\end{array}
\qquad
\begin{array}{rcl}
\mathrm{PAT} \in \textbf{Pat} & = & \$x \mid \_ \\
& \mid & M \\
& \mid & (\overrightarrow{\mathrm{PAT}}) \\
& \mid & \texttt{observe}\ \mathrm{PAT} \mid m(\overrightarrow{\mathrm{PAT}}) \\
\mathrm{SK} \in \textbf{SetCons} & = & \star \\
& \mid & M \\
& \mid & (\overrightarrow{\mathrm{SK}}) \\
& \mid & \texttt{observe}\ \mathrm{SK} \mid m(\overrightarrow{\mathrm{SK}}) \\
\eta \in \textbf{ErrorTag} & = & \texttt{prim} \mid \texttt{h-o} \mid \texttt{inf}
\end{array}
$$

Fig. 6. The syntax of $\lambda_{ds}$.

The capabilities of the two interfaces are the same; rather than issue an explicit `quit`, actors in $\lambda_{ds}$ signal termination by raising an `error`, triggering their removal from the dataspace.

A set constructor SK describes assertions, ranging from singletons to infinite sets ($\star$); $\{\ \overrightarrow{\mathrm{SK}}\ \}$ translates to an assertion set $\pi$ (from Figure 7) for placement in the dataspace. When an SK expression yields a value not suitable for dataspace assertions, such as a function, the reduction ends in an `error`. The different variants of assertions employed in a dataspace program correspond to a set of message constructors $m$, which otherwise behave like tuples.

The `project` form is the key mechanism for de-structuring incoming assertion sets in a behavior function. Specifically,

$$\texttt{project}\ \pi\ \texttt{with}\ \mathrm{PAT}\ \texttt{in}\ M$$

instantiates $M$ with the bindings of PAT for each matching assertion in $\pi$ and assembles the results into a list. A pattern's bindings may match an infinite number of values, as in

$$\texttt{project}\ \{\star\}\ \texttt{with}\ \$x\ \texttt{in}\ x$$

Our semantics interprets such expressions as errors.

### 4.1.1 *The reduction semantics of* $\lambda_{ds}$

The reduction semantics of $\lambda_{ds}$ is mostly conventional (Felleisen *et al.*, 2009); see Figure 7. The addition of assertions $c$ and assertion sets $\pi$ requires only a small extension over the standard call-by-value semantics. Assertion sets are created from set constructors with the metafunction *make-set*. The *project* metafunction implements projection, which eliminates assertion sets. Appendix A provides the full definition of each metafunction.

The semantics distinguishes among three sources of errors in order to characterize the soundness of our model precisely:

1. $\texttt{error}_{\texttt{prim}}$ arises from application of partial primitive operations;
2. $\texttt{error}_{\texttt{h-o}}$ arises from assertion sets containing functions or actors;
3. $\texttt{error}_{\texttt{inf}}$ arises when `project` selects an infinite subset of assertions.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Evaluation Syntax**

$$
\begin{array}{rcl}
M & = & \ldots \mid \pi \\
v \in \textbf{Val} & = & \lambda x.M \mid b \\
& \mid & (\overrightarrow{v}) \\
& \mid & \texttt{cons}\ v\ v \\
& \mid & \texttt{observe}\ v \mid m(\overrightarrow{v}) \\
& \mid & \pi \\
& \mid & v^{+}/v^{-} \\
& \mid & \texttt{actor}\ v_b\ v_s\ v_a \\
& \mid & \star \mid \$x \mid \_
\end{array}
\qquad
\begin{array}{rcl}
c \in \textbf{Assertion} & = & b \mid (\overrightarrow{c}) \mid \texttt{cons}\ c\ c \mid \texttt{observe}\ c \mid m(\overrightarrow{c}) \\
\pi \in \textbf{ASet} & = & \mathscr{P}(c) \\
\\
E \in \textbf{Ctx} & = & \Box \mid E\,M \mid v\,E \mid p\ \overrightarrow{v}\,E\,\overrightarrow{M} \mid (\overrightarrow{v}\ E\ \overrightarrow{M}) \\
& \mid & \texttt{cons}\ E\ M \mid \texttt{cons}\ v\ E \\
& \mid & \texttt{observe}\ E \mid m(\overrightarrow{v}\ E\ \overrightarrow{M}) \\
& \mid & \{\overrightarrow{v}\ E\ \overrightarrow{\text{SK}}\} \mid E/M \mid v/E \\
& \mid & \texttt{actor}\ E\ M\ M \mid \texttt{actor}\ v\ E\ M \\
& \mid & \texttt{actor}\ v\ v\ E \\
& \mid & \texttt{project}\ E\ \texttt{with}\ \text{PAT}\ \texttt{in}\ M \\
& \mid & \texttt{project}\ v\ \texttt{with}\ E\ \texttt{in}\ M
\end{array}
$$

**Evaluation**

$$
\mathit{eval}(M) = \begin{cases} v & \text{if } M \longrightarrow^{*} v \\ \texttt{error}_{\eta} & \text{if } M \longrightarrow^{*} \texttt{error}_{\eta} \end{cases}
$$

**Notions of Reduction**

$$
\begin{array}{rlll}
E[(\lambda x.M)\,v] & \longrightarrow E[M[x \leftarrow v]] & & (\beta_v) \\
E[p\ \overrightarrow{v}] & \longrightarrow E[v'] & \text{where } \delta(p, \overrightarrow{v}) = v' & (\delta) \\
E[p\ \overrightarrow{v}] & \longrightarrow \texttt{error}_{\texttt{prim}} & \text{where } \delta(p, \overrightarrow{v}) \text{ is undefined} & (\delta\text{-error}) \\
E[\{\overrightarrow{v}\}] & \longrightarrow E[\pi] & \text{where } \mathit{make\text{-}set}(\overrightarrow{v}) = \pi & (\text{make-set}) \\
E[\{\overrightarrow{v}\}] & \longrightarrow \texttt{error}_{\texttt{h-o}} & \text{where } \mathit{make\text{-}set}(\overrightarrow{v}) \text{ is undefined} & (\text{make-set-error}) \\
E[\texttt{project}\ \pi\ \texttt{with}\ v\ \texttt{in}\ M] & \longrightarrow E[M'] & \text{where } \mathit{project}(\pi, v, M) = M' & (\text{project}) \\
E[\texttt{project}\ \pi\ \texttt{with}\ v\ \texttt{in}\ M] & \longrightarrow \texttt{error}_{\texttt{inf}} & \text{where } \mathit{project}(\pi, v, M) \text{ is undefined} & (\text{project-error})
\end{array}
$$

**Metafunctions**

| | |
|---|---|
| *make-set* | $\overrightarrow{\textbf{Val}} \longrightarrow_{\mathit{partial}} \textbf{ASet}$ |
| $\mathit{make\text{-}set}(\overrightarrow{v})$ | translates a vector of values into a dataspace representation<br>undefined if given higher-order values |
| *project* | $\textbf{ASet} \times \textbf{Val} \times \textbf{Expr} \longrightarrow_{\mathit{partial}} \textbf{Expr}$ |
| $\mathit{project}(\pi, v_p, M)$ | creates a list by replacing pattern variables from $v_p$, an evaluated pattern, in $M$ with values from matching assertions in $\pi$<br>undefined if there is an infinite number of different matches |
| $\delta$ | $\textbf{Prim} \times \overrightarrow{\textbf{Val}} \longrightarrow_{\mathit{partial}} \textbf{Val}$<br>applies a primitive; undefined in cases due to partial primitives |

Fig. 7. The essence of the formal semantics of $\lambda_{ds}$.

### *4.2 Types for dataspaces and actors*

Figure 8 extends the syntax of $\lambda_{ds}$ with simple types, giving rise to $\lambda_{ds}^{\cup}$. The primary difference is the presence of type annotations in functions, actors, dataspaces, and patterns; additionally, expressions now include `fold` $M_c\ M_n\ M_l$, a representative induction scheme for iterating over lists. Reduction of `fold` terms is standard; lists unfold to applications of $M_c$ with base case $M_n$.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rcl}
I & = & \texttt{dataspace}\ \tau_c\ M \\
\\
M & = & \ldots \\
 & | & \lambda x{:}\tau.\, M \\
 & | & \texttt{fold}\ M\ M\ M \\
 & | & \texttt{actor}\ \tau_c\ M_b\ M_s\ M_a \\
\\
\mathrm{PAT} & = & \ldots \\
 & | & \$x{:}\tau \\
\\
v & = & \ldots \\
 & | & \lambda x{:}\tau.\, M \\
 & | & \texttt{actor}\ \tau_c\ v_b\ v_s\ v_a \\
 & | & \texttt{dataspace}\ \tau_c\ v \\
 & | & \$x{:}\tau
\end{array}
\qquad
\begin{array}{rcl}
\tau,\sigma \in \textbf{Type} & = & \tau \to \tau \mid B \\
 & | & \texttt{Observe}\ \tau \mid m(\overrightarrow{\tau}) \\
 & | & (\overrightarrow{\tau}) \mid \texttt{List}\ \tau \\
 & | & \texttt{AssertionSet}\ \tau \mid \texttt{Patch}\ \tau\ \tau \\
 & | & \bigcup \overrightarrow{\tau} \\
 & | & \texttt{Actor}\ \tau \\
 & | & \star \\
 & | & \${:}\tau \mid \texttt{Discard} \\
\\
B \in \textbf{BaseTy} & = & \text{base types: } \texttt{String}, \texttt{Int}, \text{ etc.} \\
\\
\Gamma \in \textbf{Env} & = & \overrightarrow{x{:}\tau} \\
\\
\bot & \stackrel{df}{=} & \bigcup \cdot \\
\texttt{Action}\ \tau\ \sigma & \stackrel{df}{=} & (\texttt{Patch}\ \tau\ \star) \cup (\texttt{Actor}\ \sigma)
\end{array}
$$

Fig. 8. Typed syntax of $\lambda_{ds}$.

The language of types reflects the underlying expression language and adds union types for dealing with dataspace communication. A union of types is written $\bigcup \overrightarrow{\tau}$. When convenient, we use the infix notation $\tau \cup \tau$. We do not provide any elimination forms for union types. Rather, we use unions primarily for describing the contents of assertion sets and rely on the pattern supplied to `project` to discriminate the branches of a union. The definition of the *project* metafunction is the same as the one in the untyped model, relying on the structure of the pattern to find matching assertions. Therefore, the type of the pattern must predict how matching will proceed during evaluation. For example, let $M$ describe a set of assertions $\tau$, where

$$\tau = m_1(\texttt{String}) \cup m_2(\texttt{Int}) \cup m_3()$$

Using a pattern that describes only one of the message constructors, $m_2$, as in

$$\texttt{project}\ M\ \texttt{with}\ m_2(\$x : \texttt{Int})\ \texttt{in}\ x$$

focuses type checking on only those members of the union that might match the pattern, allowing the projection to ignore $m_1$ and $m_3$. In a full implementation, we also expect a discipline such as occurrence typing (Tobin-Hochstadt & Felleisen, 2008) to facilitate programming with unions.

The type `Actor` $\tau_c$ is that of actors in a dataspace with communication type $\tau_c$.

The data constructor `observe` is directly reflected as the type constructor `Observe`. Similarly, $m(\overrightarrow{\tau})$ is the type of a message constructed by $m$ with fields $\overrightarrow{\tau}$. The type $\star$ is the type of assertions created with $\star$. Thus, the expression { `observe in-room(`$\star$`, "FBI")` } has type

$$\texttt{AssertionSet}\ (\texttt{Observe in-room}(\star, \texttt{String}))$$

The type `Patch` $\tau^+\ \tau^-$ describes patches $M^+/M^-$ where $M^+$ has type `AssertionSet` $\tau^+$ and $M^-$ type `AssertionSet` $\tau^-$. Finally, $\$ : \tau$ is the type of patterns $\$x : \tau$, `Discard` is the type of _, and `Action` $\tau\ \sigma$ abbreviates (`Patch` $\tau\ \star$) $\cup$ (`Actor` $\sigma$).

### 4.3 Static semantics

The purpose of the typing rules for $\lambda^{\cup}_{ds}$ in Figure 9 is to establish three key invariants using a standard judgment. First, actors place only valid data in their dataspace, i.e., first-order sets

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$\vdash I$$

$$\frac{\texttt{flat}(\tau_c) \qquad \vdash M_{boot} : \texttt{List}\,(\texttt{Actor}\ \tau_c)}{\vdash \texttt{dataspace}\ \tau_c\ M_{boot}}\ \text{T-DATASPACE}$$

$$\Gamma \vdash M : \tau$$

$$\frac{\Gamma \vdash x : \tau \qquad \tau <: \sigma}{\Gamma \vdash x : \tau}\ \text{T-SUB}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_{behavior} : (\texttt{Patch}\ \tau_{in}\ \tau_{in}, \tau_{state}) \to (\texttt{List}\,(\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state}) \\ \Gamma \vdash M_{state} : \tau_{state} \qquad \Gamma \vdash M_{assertions} : \texttt{AssertionSet}\ \tau_{out} \\ \tau_{out} <: \tau_c \qquad \mathit{predict\text{-}routing}(\tau_{out}, \tau_c) <: \tau_{in} \qquad \texttt{flat}(\tau_c)\end{array}}{\Gamma \vdash \texttt{actor}\ \tau_c\ M_{behavior}\ M_{state}\ M_{assertions} : \texttt{Actor}\ \tau_c}\ \text{T-ACTOR}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_s : \texttt{AssertionSet}\ \tau_s \\ \Gamma \vdash \text{PAT} : \tau_p \qquad \texttt{safe}\,(\tau_s, \tau_p) \qquad \mathit{bindings}(\text{PAT}) = \Gamma' \qquad \Gamma, \Gamma' \vdash M_b : \tau_b\end{array}}{\Gamma \vdash \texttt{project}\ M_s\ \texttt{with}\ \text{PAT}\ \texttt{in}\ M_b : \texttt{List}\ \tau_b}\ \text{T-PROJECT}$$

$$\frac{\overrightarrow{\Gamma \vdash \text{SK} : \tau}}{\Gamma \vdash \{\,\overrightarrow{\text{SK}}\,\} : \texttt{AssertionSet} \bigcup \overrightarrow{\tau}}\ \text{T-SET} \qquad \frac{\pi \models \tau}{\Gamma \vdash \pi : \texttt{AssertionSet}\ \tau}\ \text{T-}\pi$$

$$\frac{}{\Gamma \vdash \texttt{error}_\eta : \tau}\ \text{T-ERROR}$$

Fig. 9. Selected elements of an actor type system.

of assertions at the specified communication type. Second, actors extract only finite subsets of assertions via `project`. Third, well-typed actors are terminating; that is, they either signal an error or terminate with a list of actions and a new private state value. This last invariant discharges an assumption of the universal soundness theorem of Garnock-Jones *et al*. (2016).

The typing rule for dataspace programs, T-DATASPACE, ensures that the communication type $\tau_c$ describes assertions with the premise $\texttt{flat}(\tau_c)$. The `flat` judgment identifies the types of basic, first-order data such as numbers, strings, tuples of numbers, and so on suitable for dataspace assertions. Its complete definition is in Appendix A. The rule also checks that $M_{boot}$, the boot actors, all safely operate with communication type $\tau_c$.

According to T-ACTOR, an actor of shape $\texttt{actor}\ \tau_c\ M_b\ M_s\ M_a$ may participate in a dataspace of type $\tau_c$ when the type of its behavior function, $M_b$, fits the template

$$(\texttt{Patch}\ \tau_{in}\ \tau_{in}, \tau_{state}) \to (\texttt{List}\,(\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state})$$

and satisfies certain conditions concerning $\tau_{in}$, $\tau_{out}$, and $\tau_c$. Specifically, all assertions produced by the behavior function, $\tau_{out}$, must be valid utterances in $\tau_c$. Next, $\tau_{in}$, the type of assertions the actor is prepared to handle, must account for the actor's interests. Intuitively, an actor must be prepared to receive all of the assertions it asks for. The type $\tau_{in}$ describes the actor's assumptions about the sets of assertions it receives, including which subsets may be infinite. The actual assertions it receives arise from dataspace routing, which matches stated interests against all current assertions. The *predict-routing* metafunction, defined in Figure 10, mirrors run-time routing in order to predict the types of assertions received by an actor. It computes the overlap between the types of interests expressed by

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rcl}
\textit{predict-routing} & : & \textbf{Type} \times \textbf{Type} \longrightarrow \textbf{Type} \\
\textit{predict-routing}(\tau_o, \tau_c) & = & \textit{strip-obs}(\tau_o) \mathbin{\widetilde{\cap}} \tau_c
\end{array}
$$

$$
\begin{array}{rcll}
\textit{strip-obs} & : & \textbf{Type} \longrightarrow \textbf{Type} & \\
\textit{strip-obs}(\texttt{Observe}\ \tau) & = & \tau & \\
\textit{strip-obs}(\star) & = & \star & \\
\textit{strip-obs}(\bigcup \overrightarrow{\tau}) & = & \bigcup \overrightarrow{\textit{strip-obs}(\tau)} & \\
\textit{strip-obs}(\tau) & = & \bot & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbin{\widetilde{\cap}} & : & \textbf{Type} \times \textbf{Type} \longrightarrow \textbf{Type} \\
\tau \mathbin{\widetilde{\cap}} \tau & = & \tau \\
\bigcup \overrightarrow{\tau} \mathbin{\widetilde{\cap}} \sigma & = & \bigcup \overrightarrow{\tau \mathbin{\widetilde{\cap}} \sigma} \\
\tau \mathbin{\widetilde{\cap}} \bigcup \overrightarrow{\sigma} & = & \bigcup \overrightarrow{\tau \mathbin{\widetilde{\cap}} \sigma} \\
\star \mathbin{\widetilde{\cap}} \tau & = & \tau \\
\tau \mathbin{\widetilde{\cap}} \star & = & \tau \\
\texttt{List}\ \tau \mathbin{\widetilde{\cap}} \texttt{List}\ \sigma & = & \texttt{List}\ (\tau \mathbin{\widetilde{\cap}} \sigma) \\
() \mathbin{\widetilde{\cap}} () & = & () \\
(\tau_1, \overrightarrow{\tau_2}) \mathbin{\widetilde{\cap}} (\sigma_1, \overrightarrow{\sigma_2}) & = & \begin{cases} \bot & \text{if } \tau_{11} <: \bot \text{ or } \tau_{22} <: \bot \\ (\tau_{11}, \overrightarrow{\sigma_{22}}) & \text{if } \tau_{22} = (\overrightarrow{\sigma_{22}}) \end{cases} \\
 & & \text{where } \tau_{11} = \tau_1 \mathbin{\widetilde{\cap}} \sigma_1, \tau_{22} = (\overrightarrow{\tau_2}) \mathbin{\widetilde{\cap}} (\overrightarrow{\sigma_2}) \\
m(\overrightarrow{\tau_n}) \mathbin{\widetilde{\cap}} m(\overrightarrow{\sigma_n}) & = & \begin{cases} m(\overrightarrow{\sigma}) & \text{if } \tau = (\overrightarrow{\sigma}) \\ \bot & \text{otherwise} \end{cases} \\
 & & \text{where } \tau = (\overrightarrow{\tau_n}) \mathbin{\widetilde{\cap}} (\overrightarrow{\sigma_n}) \\
\texttt{Observe}\ \tau \mathbin{\widetilde{\cap}} \texttt{Observe}\ \sigma & = & \begin{cases} \bot & \text{if } \tau' <: \bot \\ \texttt{Observe}\ \tau' & \text{otherwise} \end{cases} \\
 & & \text{where } \tau' = \tau \mathbin{\widetilde{\cap}} \sigma \\
\tau \mathbin{\widetilde{\cap}} \sigma & = & \bot \qquad \text{otherwise}
\end{array}
$$

Fig. 10. Key support metafunctions.

an actor and the types of possible assertions in the dataspace. The assertions the actor may express interest in are exactly those prefixed by `Observe` in $\tau_{out}$; the metafunction *strip-obs* finds all such types, while the $\widetilde{\cap}$ metafunction determines the type representation of the overlap between such interests and the potential assertions in the dataspace, $\tau_c$ (Figure 10). Since $\star$ stands for all possible assertions, including `observe`-prefixed ones, *strip-obs* treats $\star$ as if it were the unfolding `Observe` $\star$. Finally, the second parameter of `Action` $\tau_{out}$ $\tau_c$ in T-ACTOR requires the type of any spawned actor to conform to communication type $\tau_c$.

Rule T-PROJECT eliminates two potential problems from matching a pattern with type $\tau_p$ against a set of type `AssertionSet` $\tau_s$. First, binding variables in the pattern may have an infinite number of matches. Second, matching assertions may be incompatible with the type associated with binding variables, $\$ : \tau$. The `safe` $(\tau_s, \tau_p)$ judgment enforces this constraint by finding the portion of the analyzed assertion set that may match and bind variables in the pattern. Both aspects are checked by rule PS-CAPTURE (Figure 11), which

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

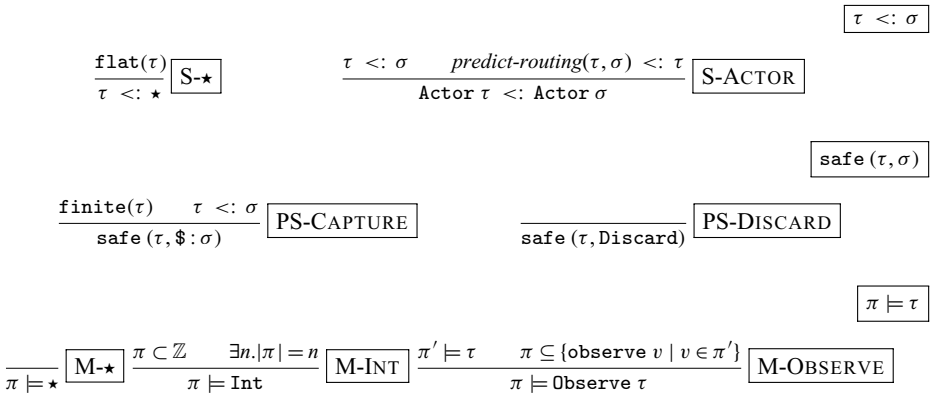


Fig. 11. Key support judgments.

corresponds to an identified match. The `finite`($\tau$) premise ensures matched assertion types do not contain $\star$, while $\tau$ <: $\sigma$ checks that they meet the pattern's expectations. Appendix A provides the full definition of each judgment.

The T-$\pi$ rule describes assertion sets $\pi$, which are not a part of the surface syntax of $\lambda^{\cup}_{ds}$. They arise through the evaluation of $\{ \overrightarrow{SK} \}$ forms and via dataspace event dispatch. Since we wish to employ a standard progress-and-preservation proof technique, we must assign them types. The $\pi \models \tau$ judgment (Figure 11; Appendix A) checks that the set $\pi$ corresponds to the structure of type $\tau$ in a way that allows it to be used by $\lambda^{\cup}_{ds}$ actors.

For example, consider the type for interest in presence of a chat room:

$$\tau_q = \texttt{Observe}\,(\texttt{in-room}(\star, \texttt{String}))$$

A set $\pi$ has type `AssertionSet` $\tau_q$ under these conditions:

- Every element of $\pi$ must be a message constructed with `in-room` holding two fields, prefixed by the `observe` constructor.
- The set of values appearing in the second slot of each message must be a finite set of strings. That is, $\{ v_2 \mid \texttt{observe in-room}(v_1, v_2) \in \pi \}$ is a finite set of `Strings`.
- The set of values appearing in the first slot of each message may be any set of assertions, including infinite sets.

The subtyping judgment `Actor` $\tau$ <: `Actor` $\sigma$ ensures that every utterance in $\tau$ is a valid utterance in $\sigma$. Furthermore, the judgment must also check for the possibility of interference. That is, transplanting the $\tau$-typed actor's subscriptions to the new $\sigma$-typed dataspace must not result in assertions that the actor is unprepared to handle. As in T-ACTOR, this condition is checked by computing *predict-routing*($\tau, \sigma$), a type that reflects routing in dataspaces.

### *4.4 Properties*

The typed language of dataspace actors satisfies a number of critical properties, most importantly, type soundness and termination.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Lemma 1** (Progress). *If* $\vdash M : \tau$ *and* $M \neq \texttt{error}_\eta$, *then either* $M \in \textbf{Val}$ *or there exists an* $M'$ *such that* $M \longrightarrow M'$.

*Proof* By case analysis on the shape of $M$. □

**Lemma 2** (Preservation). *If* $\vdash M : \tau$ *and* $M \longrightarrow M'$, *then* $\vdash M' : \tau$.

*Proof* By case analysis on possible reductions $M \longrightarrow M'$. The proof relies on several auxiliary lemmas about operations on assertion sets. In particular, Lemmas 3 and 4 show that creating assertion sets never causes an error and uses of `project` always select finite subsets of assertions. □

**Lemma 3** (Soundness of Creating Assertion Sets). *If for some* $\overrightarrow{v}$ *and* $\tau$, $\vdash \{\, \overrightarrow{v} \,\} :$ `AssertionSet` $\tau$, *then there exists* $\pi$ *such that make-set*$(\overrightarrow{v}) = \pi$ *and* $\pi \models \tau$.

*Proof* By induction on the typing derivation. □

**Lemma 4** (Soundness of `project`). *If* $\Gamma \vdash$ `project` $\pi$ `with` $v_p$ `in` $M : \tau$, *there exists an* $M'$ *such that project*$(\pi, v_p, M) = M'$, $\Gamma \vdash M' : \tau$.

*Proof* By induction on the typing derivation. □

**Theorem 5** (Soundness & Termination). *If* $\vdash M : \tau$ *and* $\texttt{error}_\eta \notin M$, *then either*

1. $M \longrightarrow^* v$ *and* $\vdash v : \tau$; *or*
2. $M \longrightarrow^* \texttt{error}_{\texttt{prim}}$.

*Interpretation* The second case of Theorem 5 implies that errors may only arise due to the application of partial primitives, never through malformed communication ($\texttt{error}_{\texttt{h-o}}$) or touching infinite sets of assertions ($\texttt{error}_{\texttt{inf}}$). □

*Proof* Starting from a well-typed term, we employ the usual progress (Lemma 1) and preservation (Lemma 2) lemmas (Wright & Felleisen, 1994) to show soundness.

To show that reduction sequences terminate with a value, we use the standard "candidate" technique (Girard, 1971). One salient detail of the proof is that, by the nature of assertions, the pattern variables in `project` are always instantiated with first-order values. □

The reduction semantics of $\lambda^{\cup}_{ds}$ programs suffices to describe how an individual actor's behavior function computes a response to an event. It does not explain, however, any properties of *dataspace actors* specified with $\lambda^{\cup}_{ds}$, that is, the meaning of a complete program `dataspace` $\tau_c$ $M$. In order to understand how systems of actors interact, we must examine the specifics of dataspace coordination, allowing us to pose and answer questions concerning actor behavior with the proper context.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rrcl}
\text{Programs} & P \in \mathbf{Prog} & = & \mathsf{dataspace}\ \vec{S} \\
\text{Actor Specifications} & S \in \mathbf{Specs} & = & \mathsf{actor}\, f\ v\ \pi \\
\text{Behavior Functions} & f \in \mathbf{BehFun} & = & \mathbf{Event} \times \mathbf{Val} \longrightarrow_{total} \overrightarrow{\mathbf{Action}} \times \mathbf{Val} + \mathbf{Error} \\
\text{Events} & e \in \mathbf{Event} & = & \Delta \\
\text{Actions} & a \in \mathbf{Action} & = & \Delta \mid S \\
\text{Patches} & \Delta \in \mathbf{Patch} & = & \pi^+/\pi^- \qquad \text{where } \pi^+ \cap \pi^- = \emptyset \\
\text{Errors} & \dagger \in \mathbf{Error} & = & \mathtt{error}_{\mathtt{ds}} \\
\text{Assertion sets} & \pi \in \mathbf{ASet} & & \text{(defined in Figure 7)} \\
\text{Assertions} & c \in \mathbf{Assertion} & & \text{(defined in Figure 7)} \\
\text{Values} & v \in \mathbf{Val} & & \text{(defined in Figure 7)}
\end{array}
$$

Fig. 12. Syntax of dataspaces.

## 5 The complete dataspace model

Dataspace coordination permits groups of actors to share knowledge via a common repository, events, and actions. Events arise when new information appears and is relevant to a stated interest of an actor. Section 5.1 gives a formal model of dataspace coordination. The model abstracts over the details of how an individual actor responds to an event, calling only for a total function that transforms a current state value to a new state and some actions. Implementing actor behavior with a specific language, such as $\lambda^{\cup}_{ds}$, requires showing that it lives up to the expected interface. It is imperative that the internal actor language and dataspace coordination model share a common ontology, enabling them to exchange (representations of) events and actions. Section 5.2 illustrates how to link the two parts of the model. Section 5.3 shows that both models meet each others' expectations, thus enjoying the desired soundness property.

### *5.1 Recap of the dataspace syntax and semantics*

Figure 12 provides a syntax for the dataspace part of our model. The internal behavior of each actor is a mathematical function. This interface is an abstraction point; it separates the concerns of an individual actor implementation from the details of dataspace concurrency and coordination. Consequently, we can just as easily use the dataspace configuration syntax and semantics with $\lambda^{\cup}_{ds}$, a model of a Racket-like functional language, as with an object-oriented model based on our ECMAScript prototype. To minimize bookkeeping, the grammar reuses the syntax of events, values, and actions from Section 4.

Writing down the reduction semantics for dataspace programs—modulo the semantics for internal actor behavior—requires three elements: (1) a generalized syntax to specify intermediate configurations; (2) several basic notions of reduction; and (3) metafunctions to keep the formulation of the reductions concise.

**Evaluation Syntax.** Figure 13 defines the evaluation syntax of dataspace programs. A running dataspace configuration $C$ contains a pending action queue, the set of all current assertions, and the contained actors. Each actor is represented by an internal name $\ell$ and a state $\Sigma$. The state of an actor consists of a queue of events to handle, a behavior

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rrcl}
\text{Configurations} & C \in \mathbf{Config} & = & [\overrightarrow{(k,a)}\,;\ R\,;\ \overrightarrow{A}] \\
\text{Behaviors} & B \in \mathbf{ABehav} & = & (f, v) \\
\text{Actors} & A \in \mathbf{Actor} & = & \ell \mapsto \Sigma \\
\text{Actor States} & \Sigma \in \mathbf{AState} & = & \overrightarrow{e} \triangleright B \triangleright \overrightarrow{a} \\
\text{Assertion Tables} & R \in \mathbf{DS} & = & \mathscr{P}(\mathit{Lift}(\mathbf{Loc}) \times \mathbf{ASet}) \\
\text{Local Labels} & j, k, \ell \in \mathbf{Loc} & = & \mathscr{N}
\end{array}
$$

Quiescent Terms

$$
\begin{array}{rcl}
A_Q \in \mathbf{Actor}_Q & = & \ell \mapsto \Sigma_Q \\
\Sigma_Q \in \mathbf{AState}_Q & = & \overrightarrow{e} \triangleright B \triangleright \cdot
\end{array}
$$

$$
\mathsf{boot} : \mathbf{Specs} + \mathbf{Prog} \to \mathbf{AState} + \mathbf{Config}
$$

$$
\mathsf{boot}\,(\mathsf{actor}\, f\ v\ \pi) = \cdot \triangleright (f, v) \triangleright \pi / \emptyset
$$

$$
\mathsf{boot}\,(\mathsf{dataspace}\ \overrightarrow{S}) = [\overrightarrow{(0, S)}\,;\ \emptyset\,;\ \cdot]
$$

Inert Terms

$$
\begin{array}{rcl}
C_I \in \mathbf{Config}_I & = & [\cdot\ ;\ R\,;\ \overrightarrow{A_I}] \\
 & \mid & (f, v) \\
A_I \in \mathbf{Actor}_I & = & \ell \mapsto \Sigma_I \\
\Sigma_I \in \mathbf{AState}_I & = & \cdot \triangleright B \triangleright \cdot
\end{array}
$$

$$
\text{Evaluation Contexts} \quad E^{\Sigma} = [\,\cdot\,;\ R\,;\ \overrightarrow{A_I}(\ell \mapsto \square)\overrightarrow{A_Q}]
$$

Fig. 13. Evaluation syntax and inert and quiescent terms of dataspaces.

$$
\overrightarrow{e}\, e_0 \triangleright (f, v) \triangleright \overrightarrow{a} \longrightarrow_{\Sigma} \overrightarrow{e} \triangleright (f, v') \triangleright \overrightarrow{a'}\,\overrightarrow{a} \qquad \boxed{\text{when } f\ (e_0, v) = (\overrightarrow{a'}, v')} \qquad \text{(notify)}
$$

$$
\overrightarrow{e}\, e_0 \triangleright (f, v) \triangleright \overrightarrow{a} \longrightarrow_{\Sigma} \cdot \triangleright (\lambda e u.(\cdot, v), v) \triangleright \emptyset\ \overrightarrow{a} \qquad \boxed{\text{when } f\ (e_0, v) \in \mathbf{Error}} \qquad \text{(exception)}
$$

$$
[\overrightarrow{(k', a)}(k, S)\,;\ R\,;\ \overrightarrow{A_Q}] \longrightarrow_{ds} [\overrightarrow{(k', a)}\,;\ R\,;\ \overrightarrow{A_Q}(\ell \mapsto \mathsf{boot}(S))] \qquad \text{(spawn)}
$$

where $\ell$ distinct from $k$, every $k'$, and the labels of every $A_Q$

$$
[\overrightarrow{(k, a)}\,;\ R\,;\ \overrightarrow{A_Q}\, A_{out}(a'')\ \overrightarrow{A}] \longrightarrow_{ds} [(\ell, a'')\,\overrightarrow{(k, a)}\,;\ R\,;\ \overrightarrow{A_Q}\, A_{out}(\cdot)\ \overrightarrow{A}] \qquad \text{(gather)}
$$

where $A_{out}(a) = \ell \mapsto \overrightarrow{e'} \triangleright B \triangleright \overrightarrow{a'} a$

$$
[\overrightarrow{(k', a)}(k, \pi^{+}/\pi^{-})\,;\ R\,;\ \overrightarrow{A_Q}] \longrightarrow_{ds} [\overrightarrow{(k', a)}\,;\ R \oplus (k, \Delta')\,;\ \overrightarrow{\mathsf{bc}_{\Delta}(k, \Delta', R, A_Q)}] \qquad \text{(patch)}
$$

where $\Delta' = (\pi^{+} - \{c \mid (k, c) \in R\})/(\pi^{-} \cap \{c \mid (k, c) \in R\})$

$$
E^{\Sigma}[\Sigma_Q] \longrightarrow_{ds} \mathit{round\text{-}robin}(E^{\Sigma}, \Sigma') \qquad \text{if} \quad \Sigma_Q \longrightarrow_{\Sigma} \Sigma' \qquad \text{(schedule)}
$$

Fig. 14. Reduction semantics of dataspaces.

$B$ comprised of a function and current private state value $(f, v)$, and a queue of actions that need to be processed by the surrounding dataspace. Quiescent terms are those without pending actions; inert terms have neither pending actions nor events to handle.

**Reduction Relation.** Figure 14 presents the reduction semantics of dataspaces. The $\longrightarrow_{\Sigma}$ relation operates on individual actor states $\Sigma$, while the $\longrightarrow_{ds}$ relation describes the

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$\mathsf{bc}_\Delta \quad : \quad \mathbf{Loc} \times \mathbf{Event} \times \mathbf{DS} \times \mathbf{Actor}_Q \longrightarrow \mathbf{Actor}$$

$$\mathsf{bc}_\Delta(k,\ \pi_{add}/\pi_{del},\ R^{old},\ \ell \mapsto \vec{e} \triangleright B \triangleright \cdot) \quad = \quad \begin{cases} \ell \mapsto \Delta_{fb}\ \ \vec{e} \triangleright B \triangleright \cdot & \text{if } \ell = k \text{ and } \Delta_{fb} \neq \emptyset/\emptyset \\ \ell \mapsto \Delta_{other}\ \ \vec{e} \triangleright B \triangleright \cdot & \text{if } \ell \neq k \text{ and } \Delta_{other} \neq \emptyset/\emptyset \\ \ell \mapsto \qquad\ \ \vec{e} \triangleright B \triangleright \cdot & \text{otherwise} \end{cases}$$

where

$$\begin{array}{rcl} \Delta_{fb} & = & \{c \mid c \in \pi_{\bullet add},\ (\ell, \texttt{observe}\ c) \in R^{new}\} \cup \{c \mid c \in (\pi_\circ \cup \pi_{\bullet add} - \pi_{\bullet del}),\ \texttt{observe}\ c \in \pi_{add}\}/ \\ & & \{c \mid c \in \pi_{\bullet del},\ \ (\ell, \texttt{observe}\ c) \in R^{old}\} \cup \{c \mid c \in \pi_\circ,\ \texttt{observe}\ c \in \pi_{del}\} \\ \Delta_{other} & = & \{c \mid c \in \pi_{\bullet add},\ (\ell, \texttt{observe}\ c) \in R^{old}\}/\{c \mid c \in \pi_{\bullet del},\ (\ell, \texttt{observe}\ c) \in R^{old}\} \\ \pi_\bullet & = & \{c \mid (j, c) \in R^{old},\ j \neq k\} \\ R^{new} & = & R^{old} \oplus (\ell, \pi_{add}/\pi_{del}) \\ \pi_{\bullet add} & = & \pi_{add} - \pi_\bullet \\ \pi_\circ & = & \{c \mid (j, c) \in R^{old}\} \\ \pi_{\bullet del} & = & \pi_{del} - \pi_\bullet \end{array}$$

Fig. 15. Dataspace routing.

reduction of dataspace configurations, with the aim of reaching quiescent or even inert states:

- notify hands an event to the behavior function of an actor and records the new state plus ensuing actions;
- exception terminates an actor that raises an exception;
- spawn creates a new actor;
- gather enqueues an action for the dataspace;
- patch realizes a state-change notification; and
- schedule selects which actor to run.

**Metafunctions.** Figure 15 defines $\mathsf{bc}_\Delta$, the metafunction that implements routing and the semantics of `observe`. It computes the relevant changes to an actor with label $\ell$ based on a patch made by actor with label $k$. In the case that $\ell = k$, it must also consider the possibility that the actor's interests change as a result of the patch. In that case, the constructed patch $\Delta_{fb}$ reflects the most up-to-date interests of the actor.

Appendix A provides the definitions of two additional metafunctions:

- $\oplus$ updates an actor's assertions in a dataspace;
- *round-robin* rotates the actors in the configuration.

### *5.2 Connecting the dots*

There is still one divide left to bridge: the models of dataspaces from Section 5.1 and the language for programming individual actors from Section 4 employ different notions of actor behavior functions. On one side, dataspaces appeal to abstract mathematical functions, while on the other side $\lambda^{\cup}_{ds}$ provides concrete λ-terms, symbolic descriptions of functions given meaning through a reduction relation. This difference must be reconciled in order to combine the two semantics into a complete understanding of dataspace programs. The two boxes in Figure 14 point out the two positions where a base language of computation interfaces with the dataspace part of the overall language.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rcl}
\mathit{behavior}_{\lambda^{\cup}_{ds}} & : & \mathbf{Event} \times \mathbf{Val} \longrightarrow \overrightarrow{\mathbf{Action}} \times \mathbf{Val} + \mathbf{Error} \\
\mathit{behavior}_{\lambda^{\cup}_{ds}}(e, (v_f, v_s)) & = & \begin{cases} (\overrightarrow{\lceil v_a \rceil}, (v_f, v'_s)) & \text{if } v = (v_{list}, v'_s) \\ \texttt{error}_{\texttt{ds}} & \text{if } v = \texttt{error}_{\eta} \end{cases} \\
 & & \text{where } M_{dispatch} = v_f\ (\lfloor e \rfloor, v_s) \\
 & & \qquad v = \mathit{eval}(M_{dispatch}) \\
 & & \qquad \overrightarrow{v_a} = \mathit{seq}(v_{list})
\end{array}
$$

---

$$
\begin{array}{rcl}
\lfloor\ \rfloor & : & \mathbf{Event} \longrightarrow \mathbf{Val} \\
\lfloor \pi^+/\pi^- \rfloor & = & \pi^+/\pi^-
\end{array}
$$

---

$$
\begin{array}{rcl}
\lceil\ \rceil & : & \mathbf{Val} \longrightarrow \mathbf{Action} \\
\lceil \pi^+/\pi^- \rceil & = & \pi^+/(\pi^- - \pi^+) \\
\lceil \texttt{actor}\ \tau\ v_f\ v_s\ \pi \rceil & = & \mathsf{actor}\ \mathit{behavior}_{\lambda^{\cup}_{ds}}\ (v_f, v_s)\ \pi
\end{array}
$$

---

$$
\begin{array}{rcl}
\mathit{seq} & : & \mathbf{Val} \longrightarrow \overrightarrow{\mathbf{Val}} \\
\mathit{seq}(\texttt{nil}) & = & \cdot \\
\mathit{seq}(\texttt{cons}\ v_h\ v_t) & = & v_h, \mathit{seq}(v_t)
\end{array}
$$

Fig. 16. Mind the gap.

The *eval* function for $\lambda^{\cup}_{ds}$ (Figure 7) provides a first step for reconciling the difference between the two parts of the model. It is a mechanism for accessing $\lambda^{\cup}_{ds}$ terms as if they were mathematical functions. Its signature, though,

$$\mathit{eval} : \mathbf{Expr} \longrightarrow \mathbf{Val} + \texttt{error}$$

is incompatible with **BehFun** (Figure 12). However, we can use it to create a behavior function. Figure 16 defines $\mathit{behavior}_{\lambda^{\cup}_{ds}}$, which is suitable for $\lambda^{\cup}_{ds}$ actors. The idea is to store each actor's private state as a pair of $\lambda^{\cup}_{ds}$ values: one for the λ-term implementing the behavior and one for the actual private state. Invoking *eval* on a function application term comprised of the behavior function, the incoming event (as a $\lambda^{\cup}_{ds}$ term[7]), and the current state value yields the actor's response to the event. To reconcile the response, a $\lambda^{\cup}_{ds}$ value, and a dataspace action, Figure 16 also defines $\lceil\ \rceil$ (pronounced "lift"), a translation that relates $\lambda^{\cup}_{ds}$ values to dataspace actions. Because the syntax of events and actions in $\lambda^{\cup}_{ds}$ is largely the same as that of dataspaces, the translation is mostly straightforward. It ensures that the two sets in a patch are disjoint (n.b. Figure 12) and performs cosmetic surgery on `actor` actions, inserting $\mathit{behavior}_{\lambda^{\cup}_{ds}}$ and storing the function term in the initial state. The final metafunction from the figure, *seq*, reconciles the `cons` lists computed in $\lambda^{\cup}_{ds}$ with the syntactic sequences manipulated by the dataspace coordination model.

[7] Though the syntax of patches is the same in both models, we include $\lfloor\ \rfloor$ (pronounced "lower") to point out where such a translation would need to happen in an implementation.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$
\begin{array}{rcl}
\mathit{initialize} & : & \mathbf{Init} \longrightarrow \mathbf{Config} + \mathbf{Error} \\
\mathit{initialize}(\texttt{dataspace}\ \tau_c\ M) & = & \begin{cases} \texttt{error}_{\texttt{ds}} & \text{if } v = \texttt{error}_\eta \\ \mathsf{boot}(\mathsf{dataspace}\ \overrightarrow{S}) & \text{otherwise} \end{cases} \\
 & & \text{where } v = \mathit{eval}(M) \\
 & & \quad \overrightarrow{v_a} = \mathit{seq}(v) \\
 & & \quad \overrightarrow{S} = \overrightarrow{\lceil v_a \rceil}
\end{array}
$$

Fig. 17. $\lambda^{\cup}_{ds}$ programs as dataspaces.

**Running $\lambda^{\cup}_{ds}$ Programs.** We finally have all the tools to define the meaning of a $\lambda^{\cup}_{ds}$ program, `dataspace` $\tau_c\ M$, with the metafunction *initialize* (Figure 17).

The function evaluates the given expression, yielding a list of `actor` descriptions. The *seq* metafunction translates the `cons` list to a syntactic sequence ($\overrightarrow{\cdot}$). Each `actor` value in the resulting sequence is lifted ($\lceil\ \rceil$) to yield an initial actor action; boot-ing the dataspace containing the actor actions yields the program's starting configuration. At this point, reduction proceeds via $\longrightarrow_{ds}$.

### *5.3 Properties*

With a complete operational description of $\lambda^{\cup}_{ds}$ dataspace systems in hand, we may now show that the soundness theorem of Section 4.4 generalizes to complete programs. Critically, $\lambda^{\cup}_{ds}$ lives up to the interface imposed on actors by the semantics of Section 5.1, and dataspace routing matches the expectations encoded in the type rules of $\lambda^{\cup}_{ds}$ from Section 4.3. In conjunction with a fairness property, the communication structure of a $\lambda^{\cup}_{ds}$ program matches static expectations.

Rule T-ACTOR (Figure 9) models dataspace routing by relating the assertions of interest made by an actor to the events it receives. Hence, we need to relate an individual actor's received events and performed actions across temporally distant reductions.

The relationship crucially relies on the fundamental theorem of dataspace event dispatch: a dataspace applies an actor's behavior function to only those assertions in which the actor has expressed a prior interest.

**Theorem 6** (Soundness of Routing (Garnock-Jones, 2017, p. 64)). *If $C_0 \longrightarrow^{+}_{ds} C_n$ where $C_n$ is a configuration that is about to dispatch an event to actor $\ell$:*

$$[\,\cdot\,;\ R_n\ ;\ \overrightarrow{A_I}(\ell \mapsto \overrightarrow{e'_n}(\pi^+/\pi^-) \triangleright (f, u) \triangleright \overrightarrow{a'_n})\overrightarrow{A_Q}]$$

*then there is some $C_i, i < n$, with $C_i = [\overrightarrow{(k, a)}\ ;\ R_i\ ;\ \overrightarrow{A}]$ such that the actor has a stated interest in each delivered assertion:*

$$\{\,(\ell, \texttt{observe}\ c) \mid c \in (\pi^+ \cup \pi^-)\,\} \subseteq R_i \qquad \square$$

Next, we show that dataspace routing preserves the type associated with assertion sets.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Lemma 7** (Preservation of Types Across Dataspace Reductions)**.** *The set of assertions $\pi$ held by a dataspace formed with communication type $\tau_c$ has the property $\pi \models \tau_c$ at each reduction step.*

*Proof* The dataspace operations on assertion sets are set union, intersection, and subtraction, all of which preserve types. □

The validity of T-ACTOR follows.

**Lemma 8** (Safe Event Dispatch)**.** *If the dataspace routes a patch $\pi^+/\pi^-$ to an actor with a behavior function of type* $(\texttt{Patch}\ \tau_{in}\ \tau_{in}, \tau_{state}) \rightarrow (\texttt{List}\ (\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state})$ *then* $\vdash \pi^+/\pi^- : \texttt{Patch}\ \tau_{in}\ \tau_{in}$.

*Proof* Dataspaces compute the intersection of assertions to determine which actors to invoke on which events. The type system accounts for this intersection in rule T-ACTOR, which predicts that events delivered to the actor correspond to a type covered by the domain of the behavior function. Theorem 6 plus Lemma 7 jointly verify that the semantics of dataspace routing matches this expectation. □

The function $behavior_{\lambda^{\cup}_{ds}}$ is the main link connecting the two parts of the dataspace model; it relies on correctness lemmas for several helper functions.

**Lemma 9** (Typed Lists Translate to Typed Sequences)**.** *If* $\vdash v : \texttt{List}\ \tau$ *then* $seq(v) = \overrightarrow{v_i}$ *and* $\overrightarrow{\vdash v_i : \tau}$.

*Proof* By routine induction. □

**Lemma 10** (Typed Action Translation)**.** *If* $\vdash v : \texttt{Action}\ \tau\ \sigma$ *then there exists an a such that* $\lceil v \rceil = a$.

*Proof* Immediate from $behavior_{\lambda^{\cup}_{ds}} \in \mathbf{BehFun}$, which Lemma 11 proves. □

We may now show that typed actors are well-behaved, always yielding a suitable answer in response to a dispatched event.

**Lemma 11** (Linking)**.** $behavior_{\lambda^{\cup}_{ds}} \in \mathbf{BehFun}$.

*Proof* First, we show that the term $M_{dispatch} = (v_f\ (\lfloor e \rfloor, v_s))$ is always well-typed. Though we do not formally state this property, it is self-evident that dataspace reductions handle an actor's private state appropriately; the behavior function is always invoked with the most recently returned, or initial, value. Therefore, the second argument of $behavior_{\lambda^{\cup}_{ds}}$ is always a pair $(v_f, v_s)$. Moreover, these pairs originate from a well-typed `actor` term passed to $\lceil\ \rceil$. Also, $behavior_{\lambda^{\cup}_{ds}}$ only ever updates the second element of the pair, leaving the function term $v_f$ unchanged. We may then conclude that $v_f$ came from a well-typed `actor` term, and by inversion of T-ACTOR it has type

$$(\texttt{Patch}\ \tau_{in}\ \tau_{in}, \tau_{state}) \rightarrow (\texttt{List}\ (\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state})$$

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Since $v_s$ is either the initial state from the `actor` term or the second element of a pair returned by $v_f$, it must have type $\tau_{state}$. By Lemma 8, each delivered patch $\pi^+/\pi^-$, and consequently $\lfloor e \rfloor$, has type $\texttt{Patch}\ \tau_{in}\ \tau_{in}$. Therefore

$$\vdash M_{dispatch} : (\texttt{List}\ (\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state})$$

Second, by Theorem 5, evaluation yields either a value of a suitable type or an error. That is, either $eval(M_{dispatch}) = v$ and $\vdash v : (\texttt{List}\ (\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state})$ or $eval(M_{dispatch}) = \texttt{error}_{\texttt{prim}}$.

- Case: $eval(M_{dispatch}) = \texttt{error}_{\texttt{prim}}$. The result of $behavior_{\lambda^{\cup}_{ds}}$ is then $\texttt{error}_{\texttt{ds}}$. Since $\texttt{error}_{\texttt{ds}} \in \textbf{Error}$, $behavior_{\lambda^{\cup}_{ds}}$ yields a suitable answer.
- Case: $eval(M_{dispatch}) = v$. By inversion, $v = (v_{list}, v'_s)$ where $\vdash v'_s : \tau_{state}$ and $\vdash v_{list} : \texttt{List}\ (\texttt{Action}\ \tau_{out}\ \tau_c)$. By Lemmas 9 and 10, the function produces a vector of actions: $\lceil seq(v_{list}) \rceil \in \overrightarrow{\textbf{Action}}$. □

Technically, $behavior_{\lambda^{\cup}_{ds}}$ is *not* a total function matching the signature of **BehFun**; it is only defined on events of the expected type, and private states that are pairs of the expected shape. Figure 12 requires behavior functions be total over the *entire* $\textbf{Event} \times \textbf{Val}$ space. However, we have shown that the function is only ever invoked on the portion of the space for which it is defined. Therefore, it meets the actual requirement of the dataspace semantics of always terminating with a suitable answer in response to a dispatched event.

When all individual actors meet the interface, dataspaces enjoy a progress property.

**Lemma 12** (Progress of Dataspace Configurations (Garnock-Jones, 2017, p. 61))**.** *Dataspace configurations are either inert or may further reduce.* □

We can now show that the soundness of $\lambda^{\cup}_{ds}$ extends to complete dataspace programs: actors fail only due to partial primitives. That is, during communication, errors arise only from the interpretation of messages, not their shape.

**Theorem 13** (System Soundness)**.** *If* $\vdash \texttt{dataspace}\ \tau_c\ M$*, then either*

- $initialize(\texttt{dataspace}\ \tau_c\ M) = \texttt{error}_{\texttt{ds}}$*; or*
- $initialize(\texttt{dataspace}\ \tau_c\ M) = \Sigma$*, where*
  - $\Sigma \longrightarrow^*_{ds} \Sigma_I$*; or*
  - *for all* $\Sigma'$*,* $\Sigma \longrightarrow^*_{ds} \Sigma'$ *implies there exits* $\Sigma''$ *such that* $\Sigma' \longrightarrow_{ds} \Sigma''$

*Proof* By inversion of $\vdash \texttt{dataspace}\ \tau_c\ M$, it must be the case that

$$\vdash M : \texttt{List}\ (\texttt{Actor}\ \tau_c)$$

Since $\lambda^{\cup}_{ds}$ is sound (Theorem 5), either

- $eval(M) = \texttt{error}_{\texttt{prim}}$; or
- $eval(M) = v$ and $\vdash v : \texttt{List}\ (\texttt{Actor}\ \tau_c)$.

In the first case, $initialize(\texttt{dataspace}\ \tau_c\ M) = \texttt{error}_{\texttt{ds}}$; the program fails during start-up, an acceptable outcome. Otherwise, evaluation yields a value $v$ with a list type.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

By Lemma 9, $seq(v) = \overrightarrow{v_a}$ with $\overrightarrow{\vdash\ v_a\ :\ \texttt{Actor}\ \tau_c}$. By inversion of the type derivation, each value in the sequence is an actor action, i.e., $\overrightarrow{v_a = \texttt{actor}\ \tau_c\ v_f\ v_s\ \pi}$.

The next step translates each $\lambda^{\cup}_{ds}$ actor specification to a dataspace process, which according to Lemma 11 is compatible with the semantics of Section 5.1:

$$\lceil \texttt{actor}\ \tau_c\ v_f\ v_s\ \pi \rceil = \mathsf{actor}\ \mathit{behavior}_{\lambda^{\cup}_{ds}}\ (v_f, v_s)\ \pi$$

and boot produces an initial actor state. Finally, Lemma 12 shows that such states either reduce to inertness or without end. □

Finally, $\lambda^{\cup}_{ds}$ dataspace systems are fair. Programmers may rely on the fact that a ready actor will eventually run.

**Theorem 14** (Fairness)**.** *If an actor is non-quiescent, it will execute within a finite number of reductions.*

*Proof* The reduction semantics of dataspaces are deterministic (Garnock-Jones, 2017, p. 62) given the fixed scheduling rule. Coupled with terminating actor behaviors (Lemma 11), and a round-robin scheduling policy (Figure 14; Appendix A.1.3, Definition 15), any non-quiescent actor eventually finds itself in the hole of an evaluation context. □

## 6 Assessment

A typical type system offers a number of attractive benefits: a design guide, documentation, and IDE support. Soundness adds reliability in different ways: compatibility checking of specification (type) and implementation (code), error prevention, debugging, and optimization. Thus far, no type system captured tuple-space-style communication precisely and none with soundness proofs. While the typical type system advantages accrue to ours, the error prevention and detection aspect deserves a special assessment.

In terms of error prevention, the type system provably eliminates many standard problems and three kinds of novel faults specific to the dataspace model of actor computation. (1) Types guarantee the absence of mistakes related to the shape of exchanged data, up to the expressivity of the base type system. (2) The type system tracks the structure of infinite assertion sets and statically detects when an actor selects a possibly infinite subset. Thus, the system prevents cases where data produced by a faulty component leads to the crash of a well-behaved one. (3) The presented types eliminate the possibility of actors that diverge while handling state-change notifications, an issue that thus far had to be handled outside the language.

Our type system also permits the imposition of stringent constraints on specific actors. Technically speaking, actor subtyping allows a developer to place individualized constraints on the assertions of each actor in a program. By the soundness property, we know that every assertion an actor makes falls within the range type of its behavior function. The contrapositive of this fact is useful as well: an assertion can never be made by the actor unless explicitly allowed by the range type. Using this ability, we can verify some aspects of well-behaved actors, such as only engaging in conversations fitting their role.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Some aspects of the dataspace communication model are beyond the expressive power of our type system. While typed actors agree on a vocabulary, $\lambda^{\cup}_{ds}$ does not provide any further guarantee about dataspace exchanges. The type system checks only that assertions are safe to utter, but not that they contribute meaningfully to the conversation. Protocols may come with temporal constraints, requirements on the maximum (or minimum) number of related assertions that may exist, and restrictions on data beyond simple types—that an identifier is globally unique, a sequence number is greater than its predecessor, and so on. All of these properties are beyond the power of our structural type system.

Type systems for concurrency often prove deadlock-freedom as a primary result. Dataspace communication is completely indirect, so in one sense all $\lambda^{\cup}_{ds}$ programs are deadlock-free. However, deadlock-like scenarios may arise. Actor A waits for an assertion $x$ before responding with an assertion $y$, while actor B waits for an assertion of $y$ before making an assertion $x$. The type system of $\lambda^{\cup}_{ds}$ is no help in this situation. The types of behavior functions capture only the entire input/output possibilities of an actor. Therefore, they cannot track the dependencies between particular assertions needed to detect the error in this program. In ongoing work, we are experimenting with a domain-specific language for dataspace actors that lends itself to tracking the relationship between particular types of assertions.

Finally, we have emphasized the importance of termination for actor systems. In reality, an actor that provably finishes computing after four billion years is just as problematic as one that diverges. The truly desired property for actors is *responsiveness*. Termination is a compromise guarantee, albeit one that is particularly useful for language models. Advances in reasoning about worst-case execution bounds (Leivant, 2001; Hoffmann & Shao, 2015) may provide a solution to this aspect of dataspace actors.

**Note.** Prior work on dataspace actors emphasizes the ability to nest collections of actors. Technically, an actor may not only spawn new actors but also new dataspaces by sending a `dataspace` action. This nested dataspace operates as if it were an actor in the current one. Nested dataspaces offer a means of spatially separating actors and conversations. Distinguished constructors route assertions between parent and child dataspaces. We omit this feature from this presentation, as it neither alters the shape of actor interactions nor poses an additional challenge for the design of our type system. Appendix B provides an overview of what is needed to accommodate arbitrary nesting of dataspaces.

## 7 Implementation and experiences

Designing a type system is an exercise in trade-offs. Every guarantee places a burden on the language implementer and a restriction on its programmers; each simplification risks losing desired precision. Implementing the proposed system as an extension to the existing Racket prototype allows us to explore these choices on realistic examples within a tight design feedback loop.

In summary, we found that:

1. The type system lends itself to a straightforward implementation.
2. The restrictions are not burdensome. Typed programs may be written in much the same style as their untyped counterparts, modulo the insertion of type annotations.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

```
1  (define-typed-syntax (actor τ-c beh st0 as0) ≫
2    #:fail-unless (flat-type? #'τ-c)
3                  "Communication type must be first-order"
4    [⊢ beh ≫ beh- ⇒ (→ (Patch τ-in τ-in) τ-st
5                       (Instruction τ-st τ-out τ-act))]
6    [⊢ st0 ≫ st0- ⇐ τ-st]
7    [⊢ as0 ≫ as0- ⇐ (AssertionSet τ-out)]
8    #:fail-unless (<: #'τ-out #'τ-c)
9                  "Actor makes assertions not allowed in this dataspace"
10   #:fail-unless (<: #'(Actor τ-act)
11                     #'(Actor τ-c))
12                 "Spawned actors not allowed in this dataspace"
13   #:fail-unless (<: (∩ (strip-? #'τ-out) #'τ-c) #'τ-in)
14                 "Not prepared to handle all inputs"
15   ------------------------------------------------------------
16   [⊢ (untyped:actor beh- st0- as0-) ⇒ (Actor τ-c)])
```

Fig. 18. T-ACTOR as a Turnstile type-and-elaboration rule.

3. For the small programs in our design feedback loop, pattern matching on incoming assertions suffices to eliminate union types.

**Dataspace Types in Action.** The examples and snippets from Section 3 are written in the concrete syntax of our implementation, which extends the untyped Racket prototype. As the chat server example illustrates, the typed program in Figure 5 is largely the same as the untyped one in Figure 3. The major difference is that the programmer must write down the communication type, though even the design of untyped programs requires similar, informal documentation of the shape of assertions in the dataspace.

**Dataspace Types as Macros.** The type checker is defined across a collection of syntax transformers that employ the types-as-macros technique (Chang *et al.*, 2017). In this scheme, the expansion of a typed term produces both an elaboration—untyped syntax implementing the desired behavior—and its type. Transformers obtain the types of subterms through recursive traversals (Flatt *et al.*, 2012) and perform some analysis before finally computing their own type and elaboration. Chang's Turnstile library manages much of the required bookkeeping, providing a notation for writing rules that resembles the standard conventions of Figure 9.

Figure 18 provides a representative sample, lightly edited for presentation, from the implementation. The code defines the typed `(actor τ-c beh st0 as0)` form following the skeleton of the T-ACTOR rule from Figure 9. The body (lines 2–14) is a sequence of premises, which come in two forms. The first, `#:fail-unless`, performs error checking. On lines 2–3, the rule checks that the given annotation sensibly describes assertions. The `flat-type?` procedure implements the `flat` judgment. If the check fails, the rule signals an error using the supplied message. The second type of premise inspects the subterms of the given syntax, utilizing Turnstile's support of bidirectional checking (Pierce & Turner, 1998) to either infer (⇒) or check (⇐) types.[8] The conclusion of the rule (line 16) specifies the elaboration as the untyped `actor` form applied to the elaboration of the behavior,

[8] The `Instruction` type constructor is a convenience for accommodating both `transition` and `quit` actions.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

state, and assertion terms and synthesizes the type of the term. Neither the conclusion nor the premises mention the environment, Γ, which is created and accessed implicitly and hygienically in this framework.

## 8 Related work

The dataspace model of coordination and computation seeks to explore a mechanism for replicating knowledge among actors that sits between the complete isolationist approach of message sending on one end and the fully shared memory one on the other end. In this spirit, Gelernter's tuple spaces and the recently derived Fact Spaces have the closest resemblance to dataspaces. Hence this section compares the type systems of these latter two approaches with the one we have imposed on dataspace actors here. In short, integration of these models with typed programming languages has never provided a sound model of communication, as our system does.

We also consider approaches for traditional actor systems and other coordination models, as well as session types, a particularly well-known approach to *behavioral* typing. Behavioral type systems typically track explicit communication actions, such as sending a message along a channel. The structure of such communication actions may then be checked for consistency between components, often resulting in strong static guarantees. The underlying communication model of dataspaces, with anonymous components publishing and subscribing (Eugster *et al.*, 2003) to relevant information, lacks such explicit interaction. We therefore consider behavioral types for dataspaces a direction for future work, and so review developments that have some promise in their application to dataspaces.

**Fact Spaces and AmbientTalk.** The Fact Spaces model (Mostinckx *et al.*, 2007), and especially its prototype implementation CRIME, is similar to the dataspace model. Programs react to both the appearance and disappearance of facts from a shared repository. Reactions are programmed in a logic coordination language, computing new facts based on current facts, and recording (implicitly) the dependencies between facts and application actions.

The Fact Spaces model has been integrated with the AmbientTalk language (Van Cutsem *et al.*, 2014). AmbientTalk is a traditional Actor model (Hewitt *et al.*, 1973; Agha, 1986) language in the mold of E (Miller *et al.*, 2005). In E and AmbientTalk, objects are organized into *vats*; each vat runs its own event loop, dispatching incoming messages to the contained objects and running the corresponding method to completion. In addition to point-to-point messaging, AmbientTalk provides a publish/subscribe communication mechanism via topic (type) tags. Topic tags form a small nominal type system but provide no structural guarantees.

Fact Spaces and AmbientTalk are based on, and always implemented in, untyped languages. Consequently, little can be guaranteed about the behavior of programs ahead of time. Ambient contracts (Scholliers *et al.*, 2011) explore, among other things, protocol enforcement for AmbientTalk programs. In addition, ambient contracts provide functionality related to service discovery and recovering from peer disconnections.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Tuple Spaces.** Linda (Gelernter, 1985; Carriero *et al.*, 1994) introduced the tuple space model of coordination. Linda resembles a blackboard style system, where processes deposit messages in the shared space that are later read and removed by other processes. LIME (Murphy *et al.*, 2006) is an extension of the tuple space model where processes register *reactions*, handler functions that run on matching tuples as they appear but, notably, not as they disappear from the space.

Linda implementations with a typed base language tend to use a single `Tuple` type for describing items retrieved from the tuple space (Wyckoff *et al.*, 1998; Picco *et al.*, 2005; Javaspaces$^{TM}$ service specification, 2017). Such unityped interfaces require casting to a more specific type after read, with the potential to fail due to type mismatches. Gigaspaces (Gigaspaces, 2017) parameterize the type of tuple space operations such that reading a tuple does not require a cast, but do not associate a type with the entire space. Consequently, there is no assurance that the type is sensible, i.e., a tuple of that type can ever be put in the space. Blossom (van der Goot, 2000) is a tuple space implementation that fixes the type of the entire tuple space, but, being based on C++, is unsound.

**Actor and Actor-like Languages.** Point-to-point actor languages have been the subject of a number of different type systems and analyses. Though the differences in the underlying communication model make direct comparisons uninformative, there are some similarities that may allow the results of one to carry over to the other in a modified form.

The Conversation Calculus (Vieira *et al.*, 2008) shares notable similarities with dataspaces in its focus on multi-party interaction between anonymous components. In the conversation calculus, communication takes place within the context of a distinct, potentially nested, conversation name. A process initiates a conversation by instantiating a service name provided by another process. Much like in dataspaces, communication within a conversation is anonymous: routing finds both an attempt to send and to receive a message with the same tag within a conversation. Conversation types (Caires & Vieira, 2009) describe the sequence of messages exchanged during each instantiation of a conversation. The type system provides flexibility in how a conversation type decomposes into multiple process types, as well as how multiple process types may merge into a conversation type, while providing the guarantee of soundness and deadlock-freedom. A significant difference that prevents their immediate application to dataspaces is the persistent nature of assertions as well as multicast-by-default communication.

The recently developed mailbox calculus and its type system (de'Liguoro & Padovani, 2018) may also have an application to dataspaces and vice versa. Though expressive enough to describe different communication mechanisms, the mailbox calculus is tailored particularly to actor-like communication; the capability to receive from a mailbox is unique, while any number of different components may send to a mailbox. Types describe the potential contents of a mailbox with patterns, which take the form of commutative regular expressions. Like assertions, such patterns do not describe the identity of the underlying components, and grant a degree of agnosticism toward the multiplicity of a message. Checking mailbox types relies on computing the pattern describing a mailbox *after* a message has been received, unlike the persistent nature of assertions in a dataspace.

He *et al.* (2014) proposed a design for Typed Akka, a more traditional message-passing actor framework for Scala. Typed Akka actors specify a particular type for the messages

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

that they receive. While we emphasize union types for describing the assertions communicated between dataspace actors, Typed Akka seeks to work within the Scala type system. Consequently, it cannot utilize unions, even though they naturally express the underlying communication.[9]

Others have considered static assurance of actor isolation through types (Srinivasan & Mycroft, 2008; Clebsch *et al.*, 2015; Haller & Loiko, 2016) or verification (Summers & Müller, 2016). These approaches focus on isolating mutable heap references among actors, versus partial failure of components discussed here. The dataspace model is functional, with actors communicating strictly first-order values, obviating the need for tracking ownership of mutable heap values.

**Session Types.** Session type systems (Honda *et al.*, 1998) provide strong guarantees about communication between concurrent components. Theoretical variants are based on the $\pi$-calculus (Milner, 1999) and describe the bidirectional flow of values between each pair of processes. Multiparty session types (Honda *et al.*, 2008) cover pairwise communication within a delineated group of processes.

Over the past three decades, the $\pi$-calculus community has demonstrated that the calculus can encode many communication and coordination mechanisms. In this spirit, it is possible to encode dataspace programs in the channel-based model of the $\pi$-calculus. The most natural encoding represents the dataspace itself as a process and then links each actor via channel to this "dataspace process." This central process holds the table of current assertions and sends notifications to all connected actors of updates to their interests, receiving back a description of the actors' next actions, and so on. In short, this encoding would mimic the semantics of dataspaces (Section 5.1), with `notify` and `gather` supplanted by analogous channel send and receive operations.

Furthermore, sessions-type researchers have succeeded in designing suitable type systems for many of these $\pi$-calculus encodings. Here is a sketch of a session type system for the dataspace encoding. From the perspective of a dataspace process with communication type $\tau_c$, the protocol along each channel would resemble a function call corresponding to our signature for behavior functions:[10]

$$\texttt{!(Event } \tau_c\texttt{).?(List (Action } \tau_{out}\ \tau_c\texttt{))}$$

Where we use $\texttt{!}\tau$ to mean sending a message of type $\tau$ along the channel, $\texttt{?}\tau$ for receive, and $\tau\texttt{.}\tau$ for sequencing.

A close look reveals that a type system for an encoding is un-enlightening for the designers of typed tuple space languages and developers who use such languages. The clearest distinction concerns the routing process itself, especially how it informs the design of the type system. Our type system refines the communication types to mirror the idea that actors receive only messages that they asked for. Technically, the type of events an actor must handle, $\tau_{in}$, is related to $\tau_c$ and the type of interests in $\tau_{out}$. Plain session types obscure this connection because the dataspace process must treat every connected actor in a uniform

[9] The next version of the language, Scala 3, is scheduled to include union types (Dotty Compiler Team, 2019), opening the possibility of more expressive types for actors.

[10] In this encoding, each actor process would maintain its private state value to obviate the need for an existential quantifier.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

manner. In other words, the type system of the encoding lacks the modicum of dependency baked into our own type system.

This observation suggests the use of dependent session types (Toninho *et al.*, 2011). With dependent session types, a developer can prove facts about dataspace routing on a per-actor basis. We consider this approach less useful than our proposed type system, even if it is equally expressive in the context of a $\pi$-calculus encoding. Most critically, the type system can no longer guide the working programmer to a natural type. Instead, this burden is shifted to the programmer, who must articulate these relationships as dependent types on a per-actor basis. We consider this an excessively large burden on the aspiring dataspace programmer because the programmer would have to mentally shift back and forth between the encoding and the direct design goal. By comparison, our type system directly incorporates an abstraction of the underlying routing mechanism—embodied in the *predict-routing* metafunction—and thus expresses type constraints in the linguistic domain in which the developer actually programs.

Nevertheless, we conjecture that translating dataspaces into sessions and channels may provide ideas for expressing and checking dataspace protocols. A particularly challenging aspect will be the broadcast-like, time-enduring nature of dataspace assertions. We expect that the recent research on session types for messaging actors (Mostrous & Vasconcelos, 2011; Crafa, 2012) might be a good starting point for this line of investigation.

## 9 Conclusion

No single coordination model can perfectly express all patterns of communication. The tuple space family of models emphasizes the need for sharing information among a group of actors, a need that is often overlooked by point-to-point models, even though the latter might be able to encode the former with patterns. The dataspace model seeks to revive interest in addressing this need for sharing, and as we have argued (Section 2.2) offers a number of advantages for programming group conversations.

Tuple space languages have never featured a type system that captures the structure of exchanged data, however. As a consequence, programmers lose the advantages that types play in design, error prevention, documentation, optimization, and so on. Moreover, a lack of types governing communication can have an especially pernicious effect on actor-like systems. Without types, a simple data-format error in a message can cause the crash of an otherwise well-behaved actor, undermining the principle of fault-isolation.

This paper presents a structural type system for dataspace actors, the first such in the tuple space family. The new type system accounts for the novel communication medium and completely eliminates data-format errors from the model. Practically speaking, this step simplifies the task of actor fault detection and recovery.

Experience with a prototype implementation of the type system suggests that it supports a range of protocols and lends itself to practical use. To cope with complex actors, the type system might need additional tools for programming with unions, such as occurrence typing, and reasoning based on type-state (Strom & Yemini, 1986). Looking beyond the internal structure of actors, dataspace protocols stand to benefit from the checking of behavioral properties as well as structural ones.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Given the similarities between dataspaces and Fact Spaces, it should be straightforward to adapt the type system to the Fact Space model and its implementations. Furthermore, the type system, by construction, has a clearly defined, narrow interface to the dataspace mechanism (Section 5). Hence, we should also be able to adapt our type system for dataspaces to similar coordination systems for actors, such as tuple spaces and pub/sub layers.

## Acknowledgments

The authors thank Stephen Chang (Northeastern, UMass Boston) for his Turnstile package and comments on early drafts of this paper. The anonymous reviewers helped clarify many passages of the original submission.

## Conflicts of Interest

The research was partially supported by CISCO and NSF grants SHF 1763922 and 1518844.

## References

Gigaspaces (2017). Gigaspaces. Accessed October 19, 2017. Available at: https://www.gigaspaces.com.

Javaspaces$^{TM}$ service specification (2017). Javaspaces$^{TM}$ service specification, version 2.3. Accessed January 24, 2017. Available at: https://river.apache.org/release-doc/current/specs/html/js-spec.html.

Agha, G. (1986) *Actors: A Model of Concurrent Computation in Distributed Systems*. Massachusetts: MIT.

Armstrong, J. (1994) *Programming Erlang*. The Pragmatic Programmers.

Armstrong, J. (2003) *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD dissertation, Stockholm: Royal Institute of Technology.

Caires, L. & Vieira, H. T. (2009) Conversation types. In ESOP, pp. 285–300.

Carriero, N. J., Gelernter, D., Mattson, T. G. & Sherman, A. H. (1994) The Linda alternative to message-passing systems. *Parallel Comput.* **20**(4), 633–655.

Chang, S., Knauth, A. & Greenman, B. (2017) Type systems as macros. In POPL, pp. 694–705.

Clebsch, S., Drossopoulou, S., Blessing, S. & McNeil, A. (2015) Deny capabilities for safe, fast actors. In International Workshop on Programming Based on Actors, Agents, and Decentralized Control. AGERE! 2015.

Clocksin, W. F. & Mellish, C. S. (1981) *Programming in Prolog*. Springer.

Crafa, S. (2012) *Behavioural Types for Actor Systems*. Technical report.

de'Liguoro, U. & Padovani, L. (2018) Mailbox types for unordered interactions. In ECOOP.

Dotty Compiler Team. (2019) Union types - more details. Accessed December 13, 2019. Available at: https://dotty.epfl.ch/docs/reference/new-types/union-types-spec.html.

ECMA. (2015) *ECMA-262: ECMAScript 2015 Language Specification*, 6th ed. ECMA International.

Englemore, R. & Morgan, A. (eds). (1988) *Blackboard Systems*. Addison-Wesley.

Eugster, P. Th., Felber, P. A., Guerraoui, R. & Kermarrec, A.-M. (2003) The many faces of publish/subscribe. *ACM Comput. Surv.* **35**(2), 114–131.

Felleisen, M., Findler, R. B. & Flatt, M. (2009) *Semantics Engineering with PLT Redex*. MIT.

Flatt, M., Culpepper, R., Darais, D & Findler, R. B. (2012) Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.* **22**(2), 181–216.

Flatt, M. & PLT. (2010) *Reference: Racket*. Technical report PLT-TR-2010-1. PLT Inc. http://racket-lang.org/tr1/.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Garnock-Jones, T. (2017). *Conversational Concurrency*. PhD dissertation, Northeastern University.
Garnock-Jones, T. & Felleisen, M. (2016) Coordinated concurrent programming in Syndicate. In ESOP, pp. 310–336.
Garnock-Jones, T., Tobin-Hochstadt, S. & Felleisen, M. (2014) The network as a language construct. In ESOP, pp. 473–492.
Gelernter, D. (1985) Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*
Girard, J.-Y. (1971) Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Stud. Logic Foundat. Math.* **63**, 63–92.
Haller, P. & Loiko, A (2016) LaCasa: Lightweight affinity and object capabilities in Scala. In OOPSLA, pp. 272–291.
He, J., Wadler, P. & Trinder, P. (2014) Typecasting actors: From Akka to TAkka. In Proceedings of the Fifth Annual Scala Workshop, pp. 23–33.
Hewitt, C., Bishop, P. B., Greif, I., Smith, B. C., Matson, T. & Steiger, R. (1973) Actor induction and meta-evaluation. In POPL.
Hoffmann, J. & Shao, Z. (2015) Automatic static cost analysis for parallel programs. In ESOP, pp. 132–157.
Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In ESOP, pp. 122–138.
Honda, K., Yoshida, N. & Carbone, M. (2008) Multiparty asynchronous session types. In POPL'08, pp. 273–284.
Jeuring, J. (1995) Polytypic pattern matching. In FPCA'95, pp. 238–248.
Leivant, D. (2001) Termination proofs and complexity certification. In Theoretical Aspects of Computer Software, pp. 183–200.
Miller, M. S., Tribble, E. D. & Shapiro, J. (2005) Concurrency among strangers. In International Symposium on Trustworthy Global Computing, pp. 195–229.
Milner, R. (1999) *Communicating and Mobile Systems: The PI Calculus*. Cambridge University.
Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C. & De Meuter, W. (2007) Fact spaces: Coordination in the face of disconnection. In Proceedings of COORDINATION 2007, pp. 268–285.
Mostrous, D. & Vasconcelos, V. T. (2011) Session typing for a featherweight erlang. In COORDINATION, pp. 95–109.
Murphy, A. L., Picco, G. P. & Roman, G.-C. (2006) Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Soft. Eng. Method.* **15**(3), 279–328.
Newell, A. & Simon, H. A. (1972) *Human Problem Solving*. Prentice Hall.
Picco, G. P., Balzarotti, D. & Costa, P. (2005) LighTS: A lightweight, customizable tuple space supporting context-aware applications. In Proceedings of SAC'05, pp. 413–419.
Pierce, B. C. (1991) *Programming with Intersection Types, Union Types, and Polymorphism.* Technical report CMU-CS-91-106, Carnegie Mellon University.
Pierce, B. C. & Turner, D. N. (1998) Local type inference. In POPL, pp. 252–265.
Scholliers, C., Harnie, D., Tanter, E., De Meuter, W. & D'Hondt, T. (2011) Ambient contracts: Verifying and enforcing ambient object compositions à la carte. *Personal Ubiquitous Comput.* **15**(4), 341–351.
Srinivasan, S. & Mycroft, A. (2008) Kilim: Isolation-typed actors for Java. In ECOOP, pp. 104–128.
Strom, R. E. & Yemini, S. (1986) Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Soft. Eng.*, 157–171.
Summers, A. J. & Müller, P. (2016) Actor services: Modular verification of message passing programs. In ESOP.
Tasharofi, S., Dinges, P. & Johnson, R. E. (2013) Why do scala developers mix the actor model with other concurrency models? In ECOOP.
Tobin-Hochstadt, S. & Felleisen, M. (2008) The design and implementation of Typed Scheme. In POPL, pp. 395–406.

Toninho, B., Caires, L. & Pfenning, F. (2011) Dependent session types via intuitionistic linear type theory. In PPDP.
Van Cutsem, T., Gonzalez Boix, E., Scholliers, C., Lombide Carreton, A., Harnie, D., Pinte, K. & De Meuter, W. (2014) AmbientTalk: Programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.* **40**(3–4), 112–136.
van der Goot, R. (2000) *High Performance Linda Using a Class Library*. PhD dissertation, Erasmus University Rotterdam.
Vieira, H. T., Caires, L. & Seco, J. C. (2008) The conversation calculus: A model of service-oriented computation. In ESOP, pp. 269–283.
Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.
Wyckoff, P., McLaughry, S. W., Lehman, T. J. & Ford, D. A. (1998) T Spaces. *IBM Syst. J.* **37**(3), 454–474.

# A Appendix

## *A.1 Auxiliary metafunction definitions*

### *A.1.1 $\lambda_{ds}$ Reduction metafunctions*

The reduction semantics of $\lambda_{ds}$ (Figure 7) mentions several metafunctions that create and analyze assertion sets. This section collects their formal definitions.

**Definition A. 1.** *The make-set metafunction creates an assertion set from a vector of values:*

$$\begin{array}{rcl} \mathit{make\text{-}set} & : & \overrightarrow{\mathbf{Val}} \longrightarrow_{\mathit{partial}} \mathbf{ASet} \\ \mathit{make\text{-}set}(\overrightarrow{v}) & = & \bigcup \overrightarrow{\pi} \qquad \text{where } \overrightarrow{\pi} = \overrightarrow{\mathit{interp}(v)} \end{array}$$

**Definition A.2.** *The interp function maps $\lambda_{ds}$ values to sets of assertions:*

$$\begin{array}{rcl} \mathit{interp} & : & \mathbf{Val} \longrightarrow_{\mathit{partial}} \mathbf{ASet} \\ \mathit{interp}(\star) & = & \mathbf{Assertion} \\ \mathit{interp}(b) & = & \{b\} \\ \mathit{interp}(m(\overrightarrow{v})) & = & \{m(\overrightarrow{v'}) \mid (\overrightarrow{v'}) \in \mathit{interp}((\overrightarrow{v}))\} \\ \mathit{interp}(()) & = & \{()\} \\ \mathit{interp}((v, \overrightarrow{v'})) & = & \{(x, \overrightarrow{y}) \mid x \in \mathit{interp}(v), (\overrightarrow{y}) \in \mathit{interp}(\overrightarrow{v'})\} \\ \mathit{interp}(\texttt{cons}\ v_1\ v_2) & = & \{\texttt{cons}\ x\ y \mid x \in \mathit{interp}(v_1),\ y \in \mathit{interp}(v_2)\} \\ \mathit{interp}(\texttt{observe}\ v) & = & \{\texttt{observe}\ x \mid x \in \mathit{interp}(v)\} \end{array}$$

**Definition A.3.** *The project function analyzes assertion sets with a pattern:*

$$\begin{array}{rcl} \mathit{project} & : & \mathbf{ASet} \times \mathbf{Val} \times \mathbf{Expr} \longrightarrow_{\mathit{partial}} \mathbf{Expr} \\ \mathit{project}(\pi, v_p, M) & = & \mathit{unroll}(m) \qquad \text{if } m \text{ is finite} \\ & & \text{where } m = \{\gamma(M) \mid v \in \pi, \mathit{match}(v,\ v_p) = \gamma\} \end{array}$$

*Successful pattern matches yield substitutions:*

$$\text{Substitutions} \quad \gamma \in \mathbf{Sub} = \mathbf{Var} \longrightarrow_{\mathit{partial}} \mathbf{Val}$$

*where composition $\gamma_1 \circ \gamma_2$ is defined in the usual manner.*

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

*Pattern matching is defined in straightforward fashion:*

$$
\begin{array}{rcl}
\mathit{match} & : & \mathbf{Val} \times \mathbf{Val} \longrightarrow_{\mathit{partial}} \mathbf{Sub} \\
\mathit{match}(v,\ \$x : \tau) & = & \{(x, v)\} \\
\mathit{match}(v,\ \_) & = & \{\} \\
\mathit{match}(b,\ b) & = & \{\} \\
\mathit{match}(\texttt{observe}\ v,\ \texttt{observe}\ v_p) & = & \mathit{match}(v,\ v_p) \\
\mathit{match}(m(\overrightarrow{v_n}),\ m(\overrightarrow{v_{pn}})) & = & \mathit{match}((\overrightarrow{v_n}),\ (\overrightarrow{v_{pn}})) \\
\mathit{match}((v, \overrightarrow{v_n}),\ (v_p, \overrightarrow{v_{pn}})) & = & \mathit{match}(v,\ v_p) \circ \gamma \quad \text{where } \mathit{match}((\overrightarrow{v_n}),\ (\overrightarrow{v_{pn}})) = \gamma \\
\mathit{match}(\texttt{cons}\ v_1\ v_2,\ \texttt{cons}\ v_{p1}\ v_{p2}) & = & \mathit{match}(v_2,\ v_{p2}) \circ \gamma \quad \text{where } \mathit{match}(v_1,\ v_{p1}) = \gamma
\end{array}
$$

*The unroll function translates the set of results to a list expression:*

$$
\begin{array}{rcl}
\mathit{unroll} & : & \mathscr{P}(\mathbf{Expr}) \longrightarrow_{\mathit{partial}} \mathbf{Expr} \\
\mathit{unroll}(\emptyset) & = & \texttt{nil} \\
\mathit{unroll}(\{M\} \uplus S) & = & \texttt{cons}\ M\ \mathit{unroll}(S)
\end{array}
$$

While the unrolling operation does not specify the order of elements, we assume a fixed ordering for the selection of elements to form the given set to make the definition deterministic.

### *A.1.2 $\lambda^{\cup}_{ds}$ Complete type judgment*

The following definitions complete the type judgment for $\lambda^{\cup}_{ds}$ terms (Section 4.3), providing the rules elided from Figures 9 and 11.

**Definition A. 4.** *The following inference rules specify the complete type judgment of $\lambda^{\cup}_{ds}$ terms, $\Gamma \vdash M : \tau$, as well as the judgment $\vdash I$ for complete dataspace programs.*

$$\frac{\texttt{flat}(\tau_c) \qquad \vdash M_{boot} : \texttt{List}\ (\texttt{Actor}\ \tau_c)}{\vdash \texttt{dataspace}\ \tau_c\ M_{boot}}\ \text{T-DATASPACE}$$

$$\frac{\Gamma \vdash x : \tau \qquad \tau <: \sigma}{\Gamma \vdash x : \tau}\ \text{T-SUB}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_{behavior} : (\texttt{Patch}\ \tau_{in}\ \tau_{in}, \tau_{state}) \to (\texttt{List}\ (\texttt{Action}\ \tau_{out}\ \tau_c), \tau_{state}) \\ \Gamma \vdash M_{state} : \tau_{state} \qquad \Gamma \vdash M_{assertions} : \texttt{AssertionSet}\ \tau_{out} \\ \tau_{out} <: \tau_c \qquad \mathit{predict\text{-}routing}(\tau_{out}, \tau_c) <: \tau_{in} \qquad \texttt{flat}(\tau_c)\end{array}}{\Gamma \vdash \texttt{actor}\ \tau_c\ M_{behavior}\ M_{state}\ M_{assertions} : \texttt{Actor}\ \tau_c}\ \text{T-ACTOR}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_s : \texttt{AssertionSet}\ \tau_s \qquad \Gamma \vdash \mathrm{PAT} : \tau_p \\ \texttt{safe}\ (\tau_s, \tau_p) \qquad \mathit{bindings}(\mathrm{PAT}) = \Gamma' \qquad \Gamma, \Gamma' \vdash M_b : \tau_b\end{array}}{\Gamma \vdash \texttt{project}\ M_s\ \texttt{with}\ \mathrm{PAT}\ \texttt{in}\ M_b : \texttt{List}\ \tau_b}\ \text{T-PROJECT}$$

$$\frac{\overrightarrow{\Gamma \vdash \mathrm{SK} : \tau}}{\Gamma \vdash \{\overrightarrow{\mathrm{SK}}\} : \texttt{AssertionSet} \bigcup \overrightarrow{\tau}}\ \text{T-SET} \qquad \frac{\pi \models \tau}{\Gamma \vdash \pi : \texttt{AssertionSet}\ \tau}\ \text{T-}\pi$$

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

$$\frac{}{\Gamma \vdash \texttt{error}_\eta : \tau}\ \text{T-Error} \qquad \frac{\Gamma, x:\tau \vdash M : \sigma}{\Gamma \vdash (\lambda x:\tau.M) : \tau \to \sigma}\ \text{T-Fun}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2}\ \text{T-App} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\overrightarrow{\Gamma \vdash M : \tau}}{\Gamma \vdash m(\overrightarrow{M}) : m(\overrightarrow{\tau})}\ \text{T-Msg} \qquad \frac{\overrightarrow{\Gamma \vdash M : \tau}}{\Gamma \vdash (\overrightarrow{M}) : (\overrightarrow{\tau})}\ \text{T-Tuple}$$

$$\frac{\Gamma \vdash M_1 : \texttt{List}\ \tau \qquad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash \texttt{cons}\ M_1\ M_2 : \texttt{List}\ (\tau \cup \sigma)}\ \text{T-Cons}$$

$$\frac{\Gamma \vdash M_c : \tau_a \to \tau_l \to \tau_a \qquad \Gamma \vdash M_n : \tau_a \qquad \Gamma \vdash M_l : \texttt{List}\ \tau_l}{\Gamma \vdash \texttt{fold}\ M_c\ M_n\ M_l : \tau_a}\ \text{T-Fold}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{observe}\ M : \texttt{Observe}\ \tau}\ \text{T-Observe} \qquad \frac{}{\Gamma \vdash b : \mathbf{B}(b)}\ \text{T-Base}$$

$$\frac{\overrightarrow{\Gamma \vdash M : \tau} \qquad \Delta(p, \overrightarrow{\tau}) = \sigma}{\Gamma \vdash p\ \overrightarrow{M} : \sigma}\ \text{T-Prim}$$

The judgment employs several additional metafunctions.

**Definition A. 5.** *The* **B** *and* $\Delta$ *metafunctions assign sound types to primitive values and operations:*

$$\begin{array}{rcl} \mathbf{B} & : & \mathbf{BasicVal} \longrightarrow \mathbf{BaseTy} \\ \Delta & : & \mathbf{Prim} \times \overrightarrow{\tau} \longrightarrow_{partial} \tau \end{array}$$

**Definition A. 6.** *The bindings function extracts the type annotations from a pattern; repeated occurrences of the same identifier result in shadowing:*

$$\begin{array}{rcll} \mathit{bindings} & : & \mathbf{Pat} \longrightarrow \mathbf{Env} & \\ \mathit{bindings}(\$x:\tau) & = & x:\tau & \\ \mathit{bindings}(m(\overrightarrow{\mathrm{PAT}})) & = & \mathit{bindings}((\overrightarrow{\mathrm{PAT}})) & \\ \mathit{bindings}((\mathrm{PAT}, \overrightarrow{\mathrm{PAT}_n})) & = & \mathit{bindings}(\mathrm{PAT}_1), \mathit{bindings}((\overrightarrow{\mathrm{PAT}_n})) & \\ \mathit{bindings}(\texttt{observe PAT}) & = & \mathit{bindings}(\mathrm{PAT}) & \\ \mathit{bindings}(\texttt{inbound PAT}) & = & \mathit{bindings}(\mathrm{PAT}) & \\ \mathit{bindings}(\texttt{outbound PAT}) & = & \mathit{bindings}(\mathrm{PAT}) & \\ \mathit{bindings}(\_) & = & \cdot & \text{otherwise} \end{array}$$

The type rules also employ several auxiliary judgments.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Definition A.7.** *The judgment* $\Gamma \vdash_P \text{PAT} : \tau$ *checks patterns separately, which allows us to limit the expressions that may be used in patterns:*

$$\frac{\texttt{flat}(\tau)}{\Gamma \vdash_P (\$x : \tau) : (\$ : \tau)}\ \text{P-CAPTURE} \qquad \frac{}{\Gamma \vdash_P \_ : \texttt{Discard}}\ \text{P-DISCARD}$$

$$\frac{\Gamma \vdash M : \tau \qquad \texttt{flat}(\tau)}{\Gamma \vdash_P M : \tau}\ \text{P-EXP} \qquad \frac{\overrightarrow{\Gamma \vdash_P \text{PAT} : \tau}}{\Gamma \vdash_P (\overrightarrow{\text{PAT}}) : (\overrightarrow{\tau})}\ \text{P-TUPLE}$$

$$\frac{\overrightarrow{\Gamma \vdash_P \text{PAT} : \tau}}{\Gamma \vdash_P m(\overrightarrow{\text{PAT}}) : m(\overrightarrow{\tau})}\ \text{P-MSG} \qquad \frac{\Gamma \vdash_P \text{PAT} : \tau}{\Gamma \vdash_P \texttt{observe PAT} : \texttt{Observe}\ \tau}\ \text{P-SUB}$$

**Definition A.8.** *Similarly, the judgment* $\Gamma \vdash_{\text{SK}} \text{SK} : \tau$ *checks assertion-set creation:*

$$\frac{}{\Gamma \vdash_{\text{SK}} \star : \star}\ \text{SK-STAR} \qquad \frac{\Gamma \vdash M : \tau \qquad \texttt{flat}(\tau)}{\Gamma \vdash_{\text{SK}} M : \tau}\ \text{SK-EXP}$$

$$\frac{\overrightarrow{\Gamma \vdash_{\text{SK}} \text{SK} : \tau}}{\Gamma \vdash_{\text{SK}} (\overrightarrow{\text{SK}}) : (\overrightarrow{\tau})}\ \text{SK-PROD} \qquad \frac{\overrightarrow{\Gamma \vdash_{\text{SK}} \text{SK} : \tau}}{\Gamma \vdash_{\text{SK}} m(\overrightarrow{\text{SK}}) : m(\overrightarrow{\tau})}\ \text{SK-MSG}$$

$$\frac{\Gamma \vdash_{\text{SK}} \text{SK} : \tau}{\Gamma \vdash_{\text{SK}} \texttt{observe SK} : \texttt{Observe}\ \tau}\ \text{SK-SUB}$$

**Definition A.9.** *The judgment* $\texttt{flat}(\tau)$ *holds for types that correspond to legal assertions:*

$$\frac{}{\texttt{flat}(B)}\ \text{F-BASE} \qquad \frac{\overrightarrow{\texttt{flat}(\tau)}}{\texttt{flat}\left(\bigcup \overrightarrow{\tau}\right)}\ \text{F-UNION} \qquad \frac{\overrightarrow{\texttt{flat}(\tau)}}{\texttt{flat}\left((\overrightarrow{\tau})\right)}\ \text{F-PROD}$$

$$\frac{\overrightarrow{\texttt{flat}(\tau)}}{\texttt{flat}\left(m(\overrightarrow{\tau})\right)}\ \text{F-MSG} \qquad \frac{\texttt{flat}(\tau)}{\texttt{flat}(\texttt{List}\ \tau)}\ \text{F-LIST}$$

$$\frac{\texttt{flat}(\tau)}{\texttt{flat}(\texttt{Observe}\ \tau)}\ \text{F-OBSERVE} \qquad \frac{}{\texttt{flat}(\star)}\ \text{F-}\star$$

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Definition A.10.** *The judgment* $\texttt{finite}(\tau)$ *describes assertion types that do not contain any uses of* $\star$*:*

$$\frac{}{\texttt{finite}(B)}\ \textsc{Fin-Base} \qquad \frac{\overrightarrow{\texttt{finite}(\tau)}}{\texttt{finite}\left(\bigcup \overrightarrow{\tau}\right)}\ \textsc{Fin-Union}$$

$$\frac{\overrightarrow{\texttt{finite}(\tau)}}{\texttt{finite}\left((\overrightarrow{\tau})\right)}\ \textsc{Fin-Tuple} \qquad \frac{\overrightarrow{\texttt{finite}(\tau)}}{\texttt{finite}\left(m(\overrightarrow{\tau})\right)}\ \textsc{Fin-Msg}$$

$$\frac{\texttt{finite}(\tau)}{\texttt{finite}(\texttt{List}\ \tau)}\ \textsc{Fin-List} \qquad \frac{\texttt{finite}(\tau)}{\texttt{finite}(\texttt{Observe}\ \tau)}\ \textsc{Fin-Sub}$$

**Definition A.11.** *Rule* T-PROJECT *employs the* $\texttt{safe}(\tau, \sigma)$ *judgment to determine if projecting a pattern of type* $\sigma$ *against a set of* $\tau$*-typed assertions could yield an infinite or ill-typed result:*

$$\frac{\texttt{finite}(\tau) \qquad \tau <: \sigma}{\texttt{safe}(\tau, \$:\sigma)}\ \textsc{PS-Capture} \qquad \frac{}{\texttt{safe}(\tau, \texttt{Discard})}\ \textsc{PS-Discard}$$

$$\frac{}{\texttt{safe}(\tau, B)}\ \textsc{PS-Base} \qquad \frac{\overrightarrow{\texttt{safe}(\tau, \sigma)}}{\texttt{safe}\left(\tau, \bigcup \overrightarrow{\sigma}\right)}\ \textsc{PS-UnionR}$$

$$\frac{\overrightarrow{\texttt{safe}(\tau, \sigma)}}{\texttt{safe}\left(\bigcup \overrightarrow{\tau}, \sigma\right)}\ \textsc{PS-UnionL} \qquad \frac{\texttt{disjoint}(\tau, \sigma)}{\texttt{safe}(\tau, \sigma)}\ \textsc{PS-Disjoint}$$

$$\frac{\overrightarrow{\texttt{safe}(\tau_i, \sigma_i)}}{\texttt{safe}\left((\overrightarrow{\tau}), (\overrightarrow{\sigma})\right)}\ \textsc{PS-Tuple} \qquad \frac{\overrightarrow{\texttt{safe}(\star, \tau_i)}}{\texttt{safe}\left(\star, (\overrightarrow{\tau})\right)}\ \textsc{PS-Tuple}\star$$

$$\frac{\overrightarrow{\texttt{safe}(\tau_i, \sigma_i)}}{\texttt{safe}\left(m(\overrightarrow{\tau}), m(\overrightarrow{\sigma})\right)}\ \textsc{PS-Msg} \qquad \frac{\overrightarrow{\texttt{safe}(\star, \tau_i)}}{\texttt{safe}\left(\star, m(\overrightarrow{\tau})\right)}\ \textsc{PS-Msg}\star$$

$$\frac{\texttt{safe}(\tau, \sigma)}{\texttt{safe}(\texttt{List}\ \tau, \texttt{List}\ \sigma)}\ \textsc{PS-List} \qquad \frac{\texttt{safe}(\star, \tau)}{\texttt{safe}(\star, \texttt{List}\ \tau)}\ \textsc{PS-List}\star$$

$$\frac{\texttt{safe}(\tau, \sigma)}{\texttt{safe}(\texttt{Observe}\ \tau, \texttt{Observe}\ \sigma)}\ \textsc{PS-Sub} \qquad \frac{\texttt{safe}(\star, \tau)}{\texttt{safe}(\star, \texttt{Observe}\ \tau)}\ \textsc{PS-Sub}\star$$

*where*

$$\texttt{disjoint}(\tau, \sigma) = \tau \mathbin{\widetilde{\cap}} \sigma <: \bot$$

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

**Definition A.12.** *Assertion sets $\pi$ are given types through the judgment $\pi \models \tau$:*

$$\frac{\pi \subset \mathbb{Z} \qquad \exists n.|\pi| = n}{\pi \models \texttt{Int}}\ \text{M-NUMBER} \qquad \frac{\pi \subseteq \textbf{Assertion}}{\pi \models \star}\ \text{M-}\star$$

$$\frac{\pi \subseteq \{(\overrightarrow{x_i}) \mid \overrightarrow{x_i \in \pi_i}\} \qquad \overrightarrow{\pi_i \models \tau_i}}{\pi \models (\overrightarrow{\tau})}\ \text{M-PROD}$$

$$\frac{\pi \subseteq \{m(\overrightarrow{x_i}) \mid \overrightarrow{x_i \in \pi_i}\} \qquad \overrightarrow{\pi_i \models \tau_i}}{\pi \models m(\overrightarrow{\tau})}\ \text{M-MSG}$$

$$\frac{\pi \subseteq \{\texttt{observe}\ v \mid v \in \pi'\} \qquad \pi' \models \tau}{\pi \models \texttt{Observe}\ \tau}\ \text{M-OBSERVE}$$

$$\frac{\pi \subseteq \{\texttt{nil}\} \cup \{\texttt{cons}\ x\ y \mid x \in \pi_1, y \in \pi_2\} \qquad \pi_1 \models \tau \qquad \pi_2 \models \texttt{List}\ \tau}{\pi \models \texttt{List}\ \tau}\ \text{M-LIST}$$

$$\frac{\pi = \bigcup \pi_i \qquad \overrightarrow{\pi_i \models \tau_i}}{\pi \models \bigcup \overrightarrow{\tau}}\ \text{M-UNION}$$

*In rule* M-UNION*, the premise $\pi = \bigcup \pi_i$ refers to (semantic) set union, while the conclusion $\bigcup \overrightarrow{\tau}$ uses (syntactic) type union. Furthermore, the rule does not require that the $\pi_i$s are disjoint or non-empty.*

**Definition A.13.** *The subtyping judgment $\tau <: \sigma$ relates compatible types:*

$$\frac{}{\tau <: \tau}\ \text{S-REFL} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \text{S-TRANS} \qquad \frac{\texttt{flat}(\tau)}{\tau <: \star}\ \text{S-}\star$$

$$\frac{\tau_2 <: \sigma_2 \qquad \sigma_1 <: \tau_1}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}\ \text{S-FUN} \qquad \frac{\overrightarrow{\tau <: \sigma}}{\bigcup \overrightarrow{\tau} <: \sigma}\ \text{S-UNIONSUB}$$

$$\frac{\exists i.\tau <: \sigma_i}{\tau <: \bigcup \overrightarrow{\sigma}}\ \text{S-UNIONSUPER} \qquad \frac{\tau <: \sigma \qquad \textit{predict-routing}(\tau, \sigma) <: \tau}{\texttt{Actor}\ \tau <: \texttt{Actor}\ \sigma}\ \text{S-ACTOR}$$

$$\frac{\tau <: \sigma}{\texttt{List}\ \tau <: \texttt{List}\ \sigma}\ \text{S-LIST} \qquad \frac{\tau <: \sigma}{\texttt{AssertionSet}\ \tau <: \texttt{AssertionSet}\ \sigma}\ \text{S-SET}$$

$$\frac{\tau <: \sigma}{\texttt{Observe}\ \tau <: \texttt{Observe}\ \sigma}\ \text{S-SUB} \qquad \frac{\overrightarrow{\tau_i <: \sigma_i}}{(\overrightarrow{\tau}) <: (\overrightarrow{\sigma})}\ \text{S-PROD}$$

$$\frac{\overrightarrow{\tau_i <: \sigma_i}}{m(\overrightarrow{\tau}) <: m(\overrightarrow{\sigma})}\ \text{S-MSG}$$

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

### A.1.3 Dataspace coordination metafunctions

The reduction semantics of dataspaces (Section 5.1) employs several metafunctions.

**Definition A. 14.** *The $\oplus$ operation incorporates a patch into a dataspace's stored knowledge:*

$$\begin{array}{rcl} \oplus & : & \mathbf{DS} \times (\mathit{Lift}(\mathbf{Loc}) \times \mathbf{Patch}) \longrightarrow \mathbf{DS} \\ R \oplus (k, \pi_{add}/\pi_{del}) & = & R \cup \{(k,c) \mid c \in \pi_{add}\} - \{(k,c) \mid c \in \pi_{del}\} \end{array}$$

**Definition A.15.** *The round-robin metafunction implements a scheduling policy, rotating the actors in a configuration:*

$$\mathit{round\text{-}robin} : E^{\Sigma} \times \mathbf{AState} \longrightarrow \mathbf{Config}$$
$$\mathit{round\text{-}robin}([\,\cdot\,;\ R\,;\ \overrightarrow{A_I}(\ell \mapsto \square)\overrightarrow{A_Q}], \Sigma) = [\,\cdot\,;\ R\,;\ \overrightarrow{A_I}\,\overrightarrow{A_Q}\,(\ell \mapsto \Sigma)]$$

## B Appendix

### *B.1 Extensions for nested dataspaces*

As mentioned in Section 6, dataspaces can be nested, forming a tree structure of actors. To keep the model simple, our presentation removes this ability. However, because nested dataspaces are a useful feature, and they are present in both our typed and untyped prototypes, they warrant additional discussion.

Here, we give an outline of how to extend the type system to accommodate hierarchical dataspaces. The operational semantics of nested dataspace systems requires non-trivial machinery, but the semantics of an internal actor language like $\lambda^{\cup}_{ds}$ is largely the same. The extensions to the type system are likewise straightforward. We refer the interested reader to Garnock-Jones *et al.*'s work (2016; 2017) for the operational semantics of nested dataspace systems.

Actors communicate with one another across dataspaces using special assertion constructors. The assertion `outbound` $c$ is pertinent to the parent of the current dataspace; routing recognizes such assertions, in much the same way as it pays special note to interests `observe`, and replicates them in the parent context. Similarly, `inbound` $c$ is an assertion from the parent dataspace pertinent to the current one. Again, routing pays special attention to interest in such assertions, `observe inbound` $c$, replicating the interest `observe` $c$ in the parent context and wrapping received events with `inbound`.

To accommodate this hierarchical runtime, we must change the type syntax in three ways. The first change to $\lambda^{\cup}_{ds}$ is the addition of expression, value, and type forms for the new assertions. The second alteration changes the status of dataspaces; rather than being reserved for descriptions of complete programs, `dataspace` is now an expression form, akin to an `actor` action:

$$M = \ldots \mid \texttt{outbound}\ M \mid \texttt{inbound}\ M \mid \texttt{dataspace}\ \tau_c\ M$$

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press

Rule T-DATASPACE changes to reflect the status of dataspaces as actor actions:

$$\frac{\texttt{flat}(\tau_c) \qquad \Gamma \vdash M : \texttt{List Actor}\ \tau_c \qquad \tau_{ctx} = \widehat{ds\text{-}route}(\tau_c)}{\Gamma \vdash \texttt{dataspace}\ \tau_c\ M : \texttt{Actor}\ \tau_{ctx}}\ \boxed{\text{T-DATASPACE}}$$

The third change concerns the metafunction $\widehat{ds\text{-}route}$, which encapsulates the `outbound`- and `inbound`-sensitive routing described above. It synthesizes a type, $\tau_{ctx}$ in which the described dataspace operates. When viewed from the parent context,

- $strip\text{-}out(\tau)$ is the type of assertions produced by the nested dataspace;
- $relay\text{-}interests(\tau)$ is the type of interests it produces; and
- $strip\text{-}in(\tau)$ is the type of assertions it expects to match stated interests.

**Definition A.16.**

$$\begin{array}{rcl} \widehat{ds\text{-}route} & : & \textbf{Type} \longrightarrow \textbf{Type} \\ \widehat{ds\text{-}route}(\tau) & = & \tau_{out} \cup \tau_{in} \cup \tau_{relay} \\ & & \text{where} \quad \tau_{out} = strip\text{-}out(\tau) \\ & & \qquad\quad \tau_{in} = strip\text{-}in(\tau) \\ & & \qquad\quad \tau_{relay} = relay\text{-}interests(\tau) \end{array}$$

The *strip-out*, *strip-in*, and *relay-interests* metafunctions follow the same structure as *strip-obs* (defined in Figure 10).

**Definition A.17.**

$$\begin{array}{rcll} strip\text{-}out & : & \textbf{Type} \longrightarrow \textbf{Type} & \\ strip\text{-}out(\texttt{Outbound}\ \tau) & = & \tau & \\ strip\text{-}out(\star) & = & \star & \\ strip\text{-}out(\bigcup \overrightarrow{\tau}) & = & \bigcup \overrightarrow{strip\text{-}out(\tau)} & \\ strip\text{-}out(\tau) & = & \bot & \text{otherwise} \end{array}$$

**Definition A.18.**

$$\begin{array}{rcll} strip\text{-}in & : & \textbf{Type} \longrightarrow \textbf{Type} & \\ strip\text{-}in(\texttt{Inbound}\ \tau) & = & \tau & \\ strip\text{-}in(\star) & = & \star & \\ strip\text{-}in(\bigcup \overrightarrow{\tau}) & = & \bigcup \overrightarrow{strip\text{-}in(\tau)} & \\ strip\text{-}in(\tau) & = & \bot & \text{otherwise} \end{array}$$

**Definition A.19.**

$$\begin{array}{rcll} relay\text{-}interests & : & \textbf{Type} \longrightarrow \textbf{Type} & \\ relay\text{-}interests(\texttt{Observe}\ (\texttt{Inbound}\ \tau)) & = & \texttt{Observe}\ \tau & \\ relay\text{-}interests(\star) & = & \star & \\ relay\text{-}interests(\bigcup \overrightarrow{\tau}) & = & \bigcup \overrightarrow{relay\text{-}interests(\tau)} & \\ relay\text{-}interests(\tau) & = & \bot & \text{otherwise} \end{array}$$

All of the properties shown in Sections 4.4 and 5.3 can be adapted to the system including nested dataspaces as well.

https://doi.org/10.1017/S0956796820000246 Published online by Cambridge University Press