The Implementation and Use of a Generic Dataflow Behaviour in Erlang

Christopher Meiklejohn
Basho Technologies, Inc.
Bellevue, WA, United States
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

Abstract

We propose a new "generic" abstraction for Erlang/OTP that aids in the implementation of dataflow programming languages and models on the Erlang VM. This abstraction simplifies the implementation of "processing elements" in dataflow languages by providing a simple callback interface in the style of the `gen_server` and `gen_fsm` abstractions. We motivate the use of this new abstraction by examining the implementation of a distributed dataflow programming variant called Lasp.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Dataflow Programming, Erlang, Concurrent Programming

1. Introduction

The dream of dataflow programming [18, 19] is to simplify the act of writing declarative applications that can easily be parallelisable but do not introduce any accidental nondeterminism. Not only does dataflow programming alleviate the need for the developer to reason about the difficulties in concurrent programming: shared memory, thread-safety, reentrancy, and mutexes; dataflow programming provides a declarative syntax focused around data and control flow. By design, dataflow programs lend themselves well to analysis and optimization, preventing the developer from having to explicitly handle parallelism in a safe and correct manner.

While the power of Erlang/OTP is in its highly concurrent, shared-nothing actor system, this only increases the potential concurrency in the system making it difficult to prevent the introduction of accidental nondeterminism in computations. Erlang provides a solution for this problem with the Open Telecom Platform (OTP) "generic" abstractions.[1] These abstractions aim to simplify concurrent programming: developers author code that adheres to a specific "behaviour"[2] and these abstractions then provide a common way to supervise, manage, and reason about processing concurrent messages to a single actor.

We propose a new Erlang/OTP "generic" abstraction for dataflow programming on the Erlang VM called `gen_flow`. This abstraction allows for the arbitrary composition of stateful actors into larger dataflow computations. This abstraction is sufficiently generic to aid in the implementation of an arbitrary dataflow language in Erlang/OTP: we motivate the use of `gen_flow` through the implementation of a deterministic distributed dataflow variant called Lasp. [14, 15]

This paper contains the following contributions:

- **`gen_flow` abstraction:** We propose `gen_flow`, a new abstraction in the style of the "generic" abstractions provided by Erlang/OTP for building dataflow "processing elements."
- **Lasp integration:** We motivate the use of this new abstraction in building the Erlang-based, Lasp programmming model for distributed computing.

2. The `gen_flow` Abstraction

We first discuss an example of how the `gen_flow` abstraction can be used to build a dataflow application and then discuss its implementation.

2.1 Overview

We propose `gen_flow`, a new abstraction for representing "processing elements" in dataflow programming. This abstraction is presented in Erlang/OTP as a behaviour module with a well defined interface and set of callback functions, similar to the existing `gen_server` and `gen_fsm` abstractions provided by Erlang/OTP.

Consider the example in Figure 1. In this example, our processing graph contains three data nodes: two input sets, and one output set, and one function that is computing the intersection of the two sets. When either of the input sets change independently, the output set should be modified to reflect the change in the input.

The goal of this abstraction is to enable the trivial composition of dataflow components to build larger dataflow applications on the Erlang VM. Figure 2 provides the Erlang code to implement the design in Figure 1. The abstraction focuses around two major components: a function that will be responsible for performing the arbitrary computation given some inputs, and a list of anonymous functions that are used to derive the value of the inputs.

[1] When we refer to the Erlang/OTP "generic" abstractions we are referring to the set of behaviours provided: `gen_server`, `gen_fsm`, `gen_event` and `supervisor`.

[2] Behaviours are essentially interfaces specifying callback functions that a module must implement.

Erlang'15, September 04 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM 978-1-4503-3805-9/15/09…$15.00.
http://dx.doi.org/10.1145/2804295.2804300

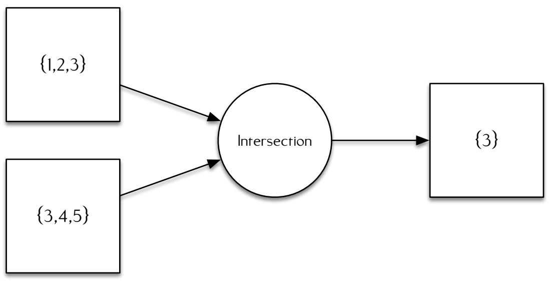

Figure 1: Dataflow example of computing the intersection of two sets. As the input sets change independently, the output is updated to reflect the change.

```erlang
-module(gen_flow_example).
-behaviour(gen_flow).

-export([start_link/1]).
-export([init/1, read/1, process/2]).

-record(state, {pid}).

start_link(Args) ->
    gen_flow:start_link(?MODULE, Args).

init([Pid]) ->
    {ok, #state{pid=Pid}}.

read(State) ->
    ReadFuns = [
        fun(_) -> sets:from_list([1,2,3]) end,
        fun(_) -> sets:from_list([3,4,5]) end
    ],
    {ok, ReadFuns, State}.

process(Args, #state{pid=Pid}=State) ->
    case Args of
        [undefined, _] ->
            ok;
        [_, undefined] ->
            ok;
        [X, Y] ->
            Set = sets:intersection(X, Y),
            Pid ! {ok, sets:to_list(Set)},
            ok
    end,
    {ok, State}.
```

Figure 2: Example use of the `gen_flow` behaviour. This module is initialized with a process identifier in the `init` function, spawns two functions to read the inputs to the `process` function, via the `read` function. The `process` function sends a message to the pid once both `ReadFuns` have returned an initial value.

2.2 Behaviour

The `gen_flow` behaviour requires three callback functions (as depicted in Figure 2):

- `Module:init/1`: Initializes and returns state.
- `Module:read/1`: Function defining how to issue requests to read inputs; takes the current state and returns a list of `ReadFuns` along with an updated state. `ReadFuns` should be arity 1 functions that take the previously read, or cached, value.
- `Module:process/2`: Function defining how to process a request; takes the current state and an argument list of the values read, and returns the new state.

In our example, `Module:init/1` is used to initialize state local to the process: this state should be used in the same fashion that the local state is used by `gen_server` and `gen_fsm`. Here, the local state is used to track the process identifier that should receive the result of the dataflow computation. `Module:init/1` is triggered once at the start of the process.

`Module:read/1` is responsible for returning a list of `ReadFuns`: these functions are responsible for retrieving the current state of the input value. In our example, we have these functions return immediately with a value of the input. However, in a pratical application, these functions would most likely talk to another process, such as a `gen_server` or `gen_fsm`, to retrieve the current state of another dataflow element.

`Module:process/2` is called every time one of the input values becomes available. This function is called with the current local state and a list of arguments that are derived from the input values returned from `Module:read/1`. In our example, once we have one value for both inputs, we compute a set intersection and send the result via Erlang message passing.

To summarize, each instance of `gen_flow` spawns a tail-recursive process that performs the following steps:

1. Launches a process for each argument in the argument list that executes that argument's `ReadFun`. This is returned by the `Module:read/1` function. This process then waits for a response from the `ReadFun` and replies back to the coordinator the result of the value. Each of these processes are **linked to the coordinator process**, so if any of them die, the entire process chain is terminated and restarted by the supervisor.
2. As soon as the coordinator receives the first response, it updates a local cache of read values for each argument in the argument list.
3. The coordinator executes the `Module:process/2` function with the latest value for each argument in the argument list from the cache. In the event that one of the argument values is not available yet, a bottom value is used; in Erlang, the atom `undefined` is used as the bottom value.
4. Inside `Module:process/2`, the user chooses how to propagate values forward. In our example, we used normal Erlang message passing.

Figure 3 diagrams one iteration of this process. Requests are initially made, from `gen_flow`, to read the current value of the inputs; once the values are known and sent back to the `gen_flow` process, the result of the function is evaluated; finally, the result is written to some output.

This model of computation is similar to how push-based functional reactive programming languages (FRP) [20] operate. In these systems, discrete changes to data items, either referred to as **events** or the more general concept of **signals** [7], notify any "processing elements" of changes to their value which triggers changes to propagate through the graph.[3]

Specifally in push-based FRP, the topology is static and events are pushed through the graph as signals change. We can imagine the `gen_flow` `ReadFuns` as establishing a just-in-time topology: they

[3] We purposely avoid the discussion of behaviors in FRP, given our system focuses on Erlang's basic data structures, none of which observe continuous changes in value.

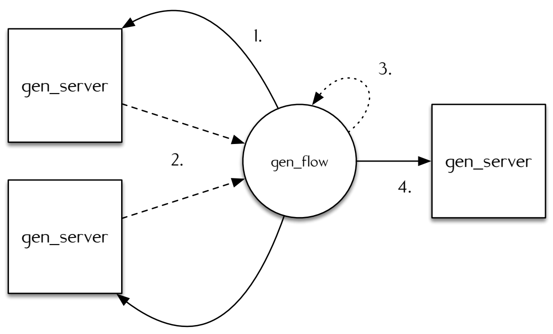

Figure 3: One iteration of the gen_flow abstraction. State is first read from inputs, computed locally, and sent to outputs. In this example, we use the gen_server abstraction for storage of state.

essentially ask the inputs to notify the **coordinator** in the event of a value change.

2.3 Reading from Inputs

When reading from input values, the gen_flow process spawns a set of linked processes to perform each read. These processes are linked to the gen_flow process, to ensure if any of them happen to fail, the entire process is crashed (and restarted, if supervised.) Additionally, each request is designated with its position in the argument list, to ensure that when responses arrive, gen_flow knows how to properly map the response from the read operation to the correct argument. We see the implementation of this in gen_flow below.

```
lists:foreach(fun(X) ->
    ReadFun = lists:nth(X, ReadFuns),
    CachedValue = orddict:fetch(X, DefaultedCache),
    spawn_link(fun() ->
                    Value = ReadFun(CachedValue),
                    Self ! {ok, X, Value}
            end)
    end,
    lists:seq(1, length(ReadFuns))).
```

For each argument to the function, a process is spawned to execute the arguments ReadFun given the previously read value taken from the local cache.

2.4 Cache

Additionally, given that the Module:process/2 function might result in the composition of the arguments, if only one input, of many, happen to change, we need to be able to recompute the function without having to retrieve a value we have already observed for the other inputs. To facilitate this, a local cache is kept at the gen_flow process and updated as the value of inputs change. This cache is maintained using an orddict local to the gen_flow process. We see the implementation of gen_flow where it performs the update of this cache and executes Module:process/2 with the most recent values below.

```
receive
    {ok, X, V} ->
        Cache = orddict:store(X, V, Cache0),
        RealizedCache = [Value || {_, Value}
                            <- orddict:to_list(Cache)],
        {ok, State} = Module:process(RealizedCache,
                                     State0)
end.
```

2.5 Usage

We envision that gen_flow can be combined with the existing "generic" abstractions provided by Erlang/OTP to build large, declarative, concurrent dataflow applications in Erlang/OTP.

Both the Erlang/OTP abstractions gen_server and gen_fsm have shown to be very powerful in practice for the management of state: gen_server representing a "generic" server process that receives and responds to messages from clients and gen_fsm representing a finite state machine that transitions based on the messages it receives.

Figure 3 outlines an example of how we imagine these facilities can be combined together. In this example, an instance of gen_flow is used to built a dataflow composition between state stored in two gen_server instances.

3. Lasp

We now motivate the use of gen_flow using Lasp.

3.1 Overview

Lasp is a distributed, fault-tolerant, dataflow programming model, with a prototypical implementation provided as a library for use in Erlang/OTP. At its core, Lasp uses distributed, convergent data structures, formalized by Shapiro et al. as Conflict-Free Replicated Data Types (CRDTs) [17], as the primary data abstraction for the developer. Lasp allows users to compose these data structures into larger applications that also observe the same properties that individual CRDTs do.

3.2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) are data structures designed for use in replicated, distributed computations. These data types come in a variety of flavors: maps, sets, counters, registers, flags, and provide a programming interface that is similar to their sequential counterparts. These data types are designed to capture concurrency properly: for example, guaranteeing deterministic convergence after concurrent additions of the same element at two different replicas of a replicated set.

One variant of these data structures is formalized in terms of bounded join-semilattices. Regardless of the type of mutation performed on these data structures and whether that function results in a change that is externally non-monotonic, state is always monotonically increasing and two states are always *join*able via a binary operation that computes a *supremum*, or least-upper-bound. To provide an example, adding to a set is always monotonic, but removing an element from a set is non-monotonic. CRDT-based sets, such as the Observed-Remove Set (OR-Set)[4] used in our example, model non-monotonic operations, such as the removal of an item from a set, in a monotonic manner. To properly capture concurrent operations that occur at different replicas of the same object, individual operations, as well as the actors that generate those operations, must be uniquely identified in the state.

The combination of monotonically advancing state, in addition to ensuring that replicas can converge via a deterministic merge operation, provides a strong convergence property: with a deterministic replica-to-replica communication protocol that guarantees that all updates are eventually seen by all replicas, multiple replicas of the same object are guaranteed to deterministically converge to the same value. Shapiro et al. have formalized this property as Strong Eventual Consistency (SEC) in [17].

To demonstrate this property, we look at three examples. In each of these examples, a circle represents an operation at a given

[4] The riak_dt_orset used in our examples is a purely functional implementation of the Observed-Remove Set (OR-Set) in Erlang.

replica and a dotted line represents a message sharing that state with another replica, where it is merged in with its current state.

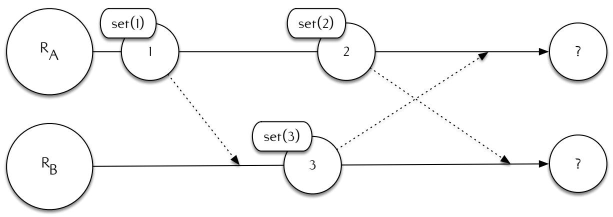

Figure 4: Example of divergence due to concurrent operations on replicas of the same object. In this case, it is unclear which update should win when replicas eventually communicate with each other.

Figure 4 diagrams an example of a distributed register. In this example, concurrent operations happen at each replica resulting in a question of how to handle the merge operation when performing replica-to-replica communication. In this example, it is up to the developer to decide how to resolve a concurrent update.

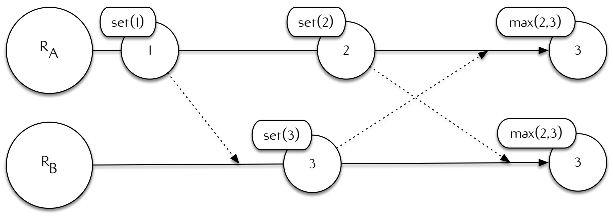

Figure 5: Example of resolving concurrent operations with a type of state-based CRDT based on a natural number lattice where the *join* operation computes *max*.

Figure 5 diagrams a simple state-based CRDT for a max value register, which extends our example in Figure 4. This data structure supports concurrent operations at each replica. In this example, concurrent operations occur where each replica sets the value to a different value (2 vs. 3). However, the CRDT ensures that the objects converge to the correct value: in this case, the **max** function, here used as the merge, is deterministic and monotonic.

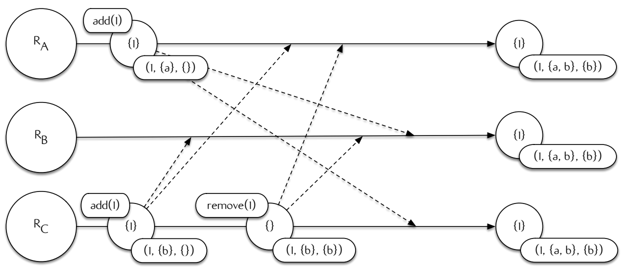

Figure 6: Example of resolving concurrent operations with an Observed-Remove Set (OR-Set). In this example, concurrent operations are represented via unique identifiers at each replica.

Finally, Figure 6 provides an example of the Observed-Remove Set (OR-Set) CRDT, a set that supports the arbitrary addition and removal of the same element repeatedly. In this set, state at each replica is represented as a set of triples, (v, a, r), where v represents the value, a is a set of unique identifiers for each addition, and r is a subset of a for each addition that has been removed. When each addition to the set is performed, the replica performing the addition generates a unique identifier for that operation; when a removal is done, the unique identifiers in the addition set are unioned into the remove set. Presence in a set for a given value is determined on whether the remove set r is a proper subset of a.

This allows the set to properly capture addition and removal operations in a monotonic fashion, supporting the removal and re-addition of the same element multiple times. One caveat does apply, however: when removing an element, removals remove all of the "observed" additions, so under concurrent additions and removals, the set biases towards additions. This is a result of attempting to provide a distributed data structure that has a sequential API. The OR-Set is just one type of CRDT that can model externally non-monotonic behaviour as monotonic growth of internal state.

Lasp is a programming model that uses CRDTs as the primary data abstraction. Lasp allows programmers to build applications using CRDTs while ensuring that the composition of the CRDTs also observed the same strong convergence properties (SEC) as the individual objects do. Lasp provides this by ensuring that the monotonic state of each object maintains a homomorphism with the program state.

3.3 API

Lasp provides five core operations over CRDTs:

- $declare(t)$: Declare a variable of type t.[5]
- $bind(x, v)$: Assign value v to variable x. If the current value of x is w, this assigns the join of v and w to x.
- $update(x, \mathrm{op}, a)$: Apply op to x identified by constant a. *op* is a data structure that performs an operation that is known to t.
- $read(x, v)$: Monotonic read operation; this operation does not return until the value of x is greater than or equal to v at which time the operation returns the current value of x.
- $strict\_read(x, v)$: Same as $read(x, v)$ except that it waits until the value of x is strictly greater than v.

Lasp provides functional programming primitives for transforming CRDT sets:

- $map(x, f, y)$: Apply function f over x into y.
- $filter(x, p, y)$: Apply filter predicate p over x into y.
- $fold(x, \mathrm{op}, y)$: Fold values from x into y using operation op.

Lasp provides set-theoretic functions for composing CRDT sets:

- $product(x, y, z)$: Compute product of x and y into z.
- $union(x, y, z)$: Compute union of x and y into z.
- $intersection(x, y, z)$: Compute intersection of x and y into z.

Figure 7 provides the code for a simple Lasp application. In this application, two sets (`S1` and `S2`) are initially created. The first set (`S1`) is updated to contain three elements, and then the values of the first set (`S1`) are composed into the second set (`S2`) via a higher-order **map** operation. This application is visually depicted in Figure 8.

Each of the functional programming primitives and set-theoretic functions are modeled as Lasp **processes**: as the values of the input CRDTs to these functions change, the value of the output CRDT resulting from applying the function also changes. Lasp processes

[5] Given the Erlang programming library does not have a rich type system, it is required to declare CRDTs with an explicit type at initialization time.

```
%% Create initial set.
{ok, S1} = lasp:declare(riak_dt_orset),

%% Add elements to initial set and update.
{ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),

%% Create second set.
{ok, S2} = lasp:declare(riak_dt_orset),

%% Apply map operation between S1 and S2.
{ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 7: Example Lasp application that defines two sets and maps the value from one into the other. We ignore the return values of the functions, given the brevity of the example.

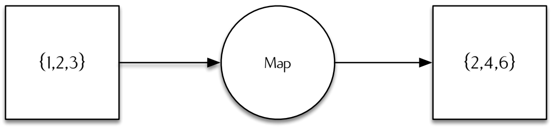

Figure 8: Dataflow graph representing the flow of information described by Figure 7.

are an instance of the gen_flow abstraction. This will be discussed in Section 3.5.

3.4 Distribution

Before discussing Lasp processes, it is important to discuss the implementation of Lasp's distributed runtime.[6]

By default, Lasp provides a centralized runtime that is used for taking single instances of CRDTs on a local machine and composing them into larger applications as discussed in Section 3.3.

In order for Lasp to support highly-available and fault-tolerant distributed computations, Lasp provides a distributed runtime for the execution of Lasp applications. The distributed runtime for Lasp ensures that variables are replicated across a cluster of nodes and operations are performed against a majority quorum of these replicas; this ensures that Lasp applications can tolerate a number of failures while still making progress. The CRDTs that Lasp uses as the primary data abstraction provide safety: even under failures and message re-orderings or duplication, computations will deterministically converge to the correct result once all messages are delivered.

If we return to Figure 8, it is important to realize that all objects in this graph are replicated: there are three copies of the input, three copies of the output, and three copies of the computation running in a distributed cluster at the same time. Additionally, each node in this graph may or may not be running on the same node in the cluster. This is visually depicted in Figure 9.

In Figure 9, it is also important to realize that some replicas may temporarily lag behind, or contain earlier values, given failures in the network. We rely on majority quorums to ensure that we can tolerate failures, in addition to an anti-entropy protocol to ensure that all replicas eventually receive all messages. Under failure conditions, Lasp operations, like **map**, may need to talk to a replica that contains earlier state.

[6] The implementation of Lasp's distributed runtime is out of scope for this paper. However, the reader is referred to our previous work [4] for a discussion on building a distributed deterministic dataflow variant on top of the Erlang-based, Riak Core [12] distributed systems framework.

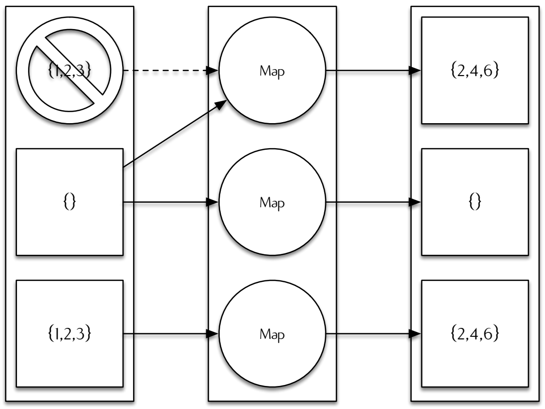

Figure 9: Replicated execution of Figure 8. In this example, since majority quorums are used to perform update operations, some replicas may lag behind until receiving state from another replica due to failures in the network. In the event of a failure, functional operations like the **map** may need to talk to a replica that contains earlier state.

3.5 Execution

As discussed in Section 3.3, all of Lasp's functional programming primitives and set-theoretic functions are implemented in terms of Lasp processes. Lasp **processes** are recursive processes that wait for changes in any of their inputs, compute a function over those inputs, and produce and output based on the functions execution. Lasp processes are implemented in terms of gen_flow.

Lasp's runtime can operate in two modes: centralized and distributed. The distributed runtime is the default for Lasp: variables are replicated across a cluster of nodes. The centralized runtime is provided for testing the semantics of Lasp independently of the distribution layer.

In the centralized runtime, Lasp's variables are stored locally in either a LevelDB instance [5] or in an ETS (Erlang Term Storage) table. The centralized runtime is used for the execution of our QuickCheck [6] model which verifies that the semantics are correct. Additionally, the centralized runtime can be used as the basis for another distribution layer.

In the distributed runtime, Lasp's variables are distributed across a cluster of nodes. Each node, because of Lasp's implementation as a Riak Core application, uses a single Erlang process for managing access to the variables at that node. In Riak Core, this process is referred to as a instance of the riak_core_vnode behaviour; this behaviour is nothing more than a wrapped gen_fsm behaviour with some additional operations focused around distribution of the state and how to route requests in the cluster.

In both of these cases, the ability to provide specific ReadFuns to gen_flow, as discussed in Section 2.2, has proven very valuable. Let us look at two examples:

Centralized Execution When testing the transformation of state with our QuickCheck model, we want to avoid routing requests through the distribution layer. The primary reasons for this are twofold: distribution adds latency to each operation, and distribution makes it harder to reason about when messages may be delivered. In this model, at compile time, we use an Erlang macro to override the ReadFun of our "processing elements" to operate locally on an ETS table and return immediately. This allows for faster execution and the ability to test language semantics separately from the distributed runtime.

Distributed Execution In the distributed execution, we have two concerns that do not appear in the centralized execution. First, replicas may become unavailable and stale replicas may be contacted, as depicted in Figure 9. In this case, we want to ensure that the functions we provide as `ReadFuns` only read forward: Lasp provides a read operation that only returns if the objects state is monotonically greater than a previously observed state; this is commonly referred to as a session guarantee in distributed systems literature. Finally, in the event that replicas may exist on remote nodes, the `ReadFuns` should contain information on how to route the request based on its location.

3.6 Example

Figure 10 shows Lasp's use of `gen_flow`. In this example, the Lasp function and its inputs are passed in through the `gen_flow` initialization function and stored in local state. The `ReadFuns` supplied use the Lasp monotonic read operation: we ensure that given the previous value observed from the cache, we always read forward in causal time. This prevents the observance of earlier values in the event of failures.

4. Evaluation

We have found that having a generic abstraction for dataflow programming has allowed us to greatly simplify the implementation of three dataflow variants: Derflow [4], Derflow$_L$ [13], and Lasp [14].

Our previous work on Derflow provides a distributed, deterministic, single-assignment dataflow programming model for Erlang that was later extended to operate over bounded join-semilattices with Derflow$_L$. Both of these models are direct precursors to Lasp and during the implementation of Lasp we were able to greatly simplify the dataflow "processing elements" by using the generic abstraction presented in this paper. This greatly reduced the complexity and code duplication of the implementation of Lasp; for example, the implementation of an operation that applies identity from one input to an output was reduced from 455 LOC to 128 LOC. Similar results exist for the other operations in Lasp. Additionally, using `gen_flow` enabled us to test the implementation of the CRDT transformation independently of the distribution layer, which was not possible with Derflow and Derflow$_L$.

5. Related Work

In the following section, we identify related work.

5.1 Kahn Process Networks

Kahn process networks (KPNs) [11] present a general model of parallel computation where processes are used to compose data that arrives on input channels into output channels. KPNs are both deterministic and monotonic and are modeled around processes that never terminate.

The `gen_flow` abstraction is both influenced by, and can be used to build applications in the style of, KPNs. Supervised instances of `gen_flow` can be used to model a computing process in a KPN; each of these instances of `gen_flow` never terminates unless instructed to by the application or terminated as a result of a fault in the system. Similar to the binding of the formal parameters at the call site where processes in a KPN are instantiated, functions for reading inputs and producing outputs with `gen_flow` can be provided at runtime or compile time, as seen in Figure 10.

5.2 Functional Reactive Programming

We acknowledge the relationship with Functional Reactive Programming [9, 20], but focus on libraries providing dataflow abstractions that can be combined with idiomatic programming in the host language instead of the traditional approach with domain specific languages.

```erlang
-module(lasp_process).
-behaviour(gen_flow).
-export([start_link/1]).
-export([init/1, read/1, process/2]).
-record(state, {read_funs, function}).

start_link(Args) ->
    gen_flow:start_link(?MODULE, Args).

%% @doc Initialize state.
init([ReadFuns, Function]) ->
    {ok, #state{read_funs=ReadFuns,
                function=Function}}.

%% @doc Return list of read functions.
read(#state{read_funs=ReadFuns0}=State) ->
    ReadFuns = [gen_read_fun(Id, ReadFun) ||
        {Id, ReadFun} <- ReadFuns0],
    {ok, ReadFuns, State}.

%% @doc Computation to execute when inputs change.
process(Args, #state{function=Function}=State) ->
    case lists:any(fun(X) -> X =:= undefined end,
         Args) of
        true ->
            ok;
        false ->
            erlang:apply(Function, Args)
    end,
    {ok, State}.

%% @doc Generate ReadFun.
gen_read_fun(Id, ReadFun) ->
    fun(Value0) ->
        Value = case Value0 of
            undefined ->
                undefined;
            {_, _, V} ->
                V
        end,
        {ok, Value1} = ReadFun(Id, {strict, Value}),
        Value1
    end.
```

Figure 10: Lasp's use of `gen_flow`. In this example, inputs and the Lasp function are passed as arguments to `gen_flow` and stored in local state. The `ReadFuns` supplied take the previously observed value from the cache and perform a monotonic read: this ensures we only ever read forward in causal time.

5.3 FlowPools

FlowPools [16] provide a lock-free, deterministic, concurrent dataflow abstraction for the Scala programming language. FlowPools are essentially a lock-free collection abstraction that support a concurrent append operation and a set of combinators and higher-order operations. FlowPools are designed for multi-threaded computation, but not distributed computation.

FlowPools have a similar abstraction to `gen_flow`, but are focused around connecting collections together using combinators. FlowPools allow lock-free append operations to be performed to add elements to the collection and on these collections support two types of functional transformations: `foreach` and `aggregate`. As elements are added, the `foreach` operation asynchronously executes and applies a transformation to a given collection: to guarantee determinism when functions provided to the foreach may contain side-effects, FlowPools ensure that the function supplied to `foreach` is only called once for each element in the collection. `aggregate` is similar to a fold in functional programming: to ensure determinism a collection must be "sealed" before completing the fold operation across the collection.

5.4 Javelin

Javelin [1] is a Clojure library for performing spreadsheet-like, cell-based dataflow programming. Javelin lets you declare "cells": a "cell" is given an arbitrary S-expression with arguments of other "cell"s. As the value of the source "cell"s change, the S-expression is re-evaluated with the current value of the inputs.

Javelin is unique in that it leverages Clojure's `IWatchable` interface: types that implement `IWatchable` execute a list of functions, stored in the object's metadata, whenever the object's value changes. Our solution uses Erlang behaviours, given that Erlang does not have a way to extend the type system in a similar fashion.

5.5 Luke and Riak Pipe

Riak Pipe [10] and its predecessor, Luke [2] are both Erlang libraries for performing "pipeline processing" developed by Basho Technologies. Both of these libraries focus around creating acyclic processing graphs and provide a similar behaviour interface.

Both of these libraries focus on fixed topologies; the system can not create a new node in the graph once the topology has been instantiated and processing has began. This is a conscious design decision; these frameworks were originally designed for MapReduce [8] style processing in Riak Core [12] based systems where a "pipeline" is established to process a finite set of data with a set group of "phases", or "processing elements." By design, if either of the "phases" happen to fail, the entire "pipeline" is collapsed and an error returned to the caller.

Riak Pipe and Luke also rely on Erlang message passing to deliver output results to the next stage of processing. Riak Pipe synchronizes on the mailbox size, through a series of acknowledgements from the receiver, to support a backpressure mechanism to prevent overloading slow "phases". This is similar to the design outlined by Welsh et al. in their work on SEDA. [21]

We believe that the `gen_flow` abstraction is generic enough to support the implementation of Riak Pipe and Luke. We plan to explore an implementation of Riak Pipe that uses `gen_flow`.

6. Conclusion

We have presented a new "generic" abstraction for Erlang/OTP to support the implementation of dataflow programming languages called `gen_flow`. We have motivated the use of this new abstraction through the implementation of a distributed, deterministic dataflow programming model called Lasp. We have demonstrated that use of this new abstraction has helped reduce code duplication and made it easier to reason about computations in dataflow programming in Erlang/OTP, as well as provided a way to integrate dataflow "processing elements" with the existing Erlang/OTP "generic" abstractions.

A. Source Code Availability

Lasp and `gen_flow` are available on GitHub under the Apache 2.0 License at `http://github.com/lasp-lang/lasp`.

Acknowledgments

Thanks to Sean Cribbs for providing valuable feedback during the research and development of this work. The research leading to these results has been partially funded by the SyncFree Project [3] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 609551.

References

[1] Javelin source code repository. `https://github.com/tailrecursion/javelin`. Accessed: 2015-05-23.

[2] Luke source code repository. `https://github.com/basho/luke`. Accessed: 2015-05-22.

[3] SyncFree: Large-scale computation without synchronisation. `https://syncfree.lip6.fr`. Accessed: 2015-02-13.

[4] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 46(4): 53–64, 2011.

[7] E. Czaplicki. Elm: Concurrent FRP for Functional GUIs. *Master's thesis, Harvard*, 2012.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2009.

[10] B. Fink. Distributed computation on Dynamo-style distributed storage: Riak Pipe. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang*, pages 43–50. ACM, 2012.

[11] G. Kahn. The semantics of a simple language for parallel programming. In *In Information Processing74: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[12] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[13] C. Meiklejohn. Eventual Consistency and Deterministic Dataflow Programming. *8th Workshop on Large-Scale Distributed Systems and Middleware*, 2014.

[14] C. Meiklejohn and P. Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195. ACM, 2015.

[15] C. Meiklejohn and P. Van Roy. Lasp: a language for distributed, eventually consistent computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[16] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A lock-free deterministic concurrent dataflow abstraction. In *Languages and Compilers for Parallel Computing*, pages 158–173. Springer, 2013.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 01 2011.

[18] T. B. Sousa. Dataflow Programming Concept, Languages and Applications. In *Doctoral Symposium on Informatics Engineering*, 2012.

[19] Van Roy, Peter and Haridi, Seif. *Concepts, techniques, and models of computer programming*. MIT Press, 2004.

[20] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Notices*, volume 35, pages 242–252. ACM, 2000.

[21] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.