Stream-oriented Robotics Programming: The Design of roshask

Anthony Cowley and Camillo J. Taylor

***Abstract*— The decomposition of robotics software into a collection of loosely coupled processes has become a core design principle of virtually every large robotics software engineering effort over the past decade. Recently, the ROS software platform from Willow Garage has gained significant traction due to its adoption of sound design principles and significant software library contributions from Willow Garage itself. This paper describes a binding from the Haskell programming language to basic ROS interfaces. The novelty of these bindings is that they allow for, and encourage, a higher level of abstraction in writing programs for robots that treat streams of values as first-class citizens. This approach makes the fusing, transforming, and filtering of streams fully generic and compositional while maintaining full compatibility with the existing ROS ecosystem.**

I. Introduction

Component oriented design has driven the robotics community for many years now. Robotic hardware is assembled from well-understood components that may be fit together to create more complex mechatronic systems, while software has followed a parallel evolutionary path in an effort to mimic that same modularity. While the design technique of decomposing a large problem into discrete sub-problems is sound engineering practice, maintaining the integrity of component abstractions requires that the abstractions engineers are asked to respect are efficient, expressive, and consistent.

The advantage of maintaining abstraction boundaries around individual components – wherein each component is a black box with an interface whose simplicity may belie internal complexity – is that the component designer addresses a modestly-scoped problem, while the systems integrator is equipped to work with large *units of functionality*, each of which has some identifiable role to play in the context of the system as a whole. The division of a system along lines governed by semantically significant separations of concerns informs both collaborative development and runtime deployment strategies.

In this paper, we extend the large units of functionality visible at the system level of ROS programming into the development of lower level software components. This approach presents advantages in terms of brevity and compactness in program code, but also vastly simplifies the expression of behaviors dependent on multiple, asynchronous data sources.

We have developed `roshask` to represent ROS topics as first-class values that may be passed to functions as arguments and returned as results. The implementation of ROS nodes is designed in a complementary fashion to specify the naming and wiring together of individual topics in a compositional manner. The approach described here enables ROS programming at a high level of abstraction with minimal overhead.

Anthony Cowley and Camillo J. Taylor are with the GRASP Laboratory at the University of Pennsylvania, Philadelphia, PA 19104, USA {acowley,cjtaylor}@seas.upenn.edu

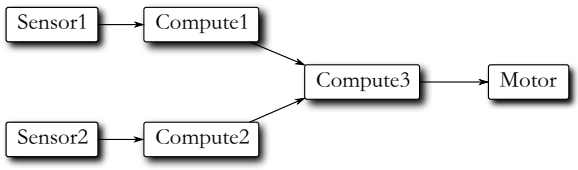

Fig. 1. Example arrangement of loosely coupled components that pass messages to effect reactive behavior. Hardware interface components feed data to processing components whose outputs are then fed to a third computational component that produces commands for a motor interface.

For example, to implement the `Compute3` node of the system shown in Figure 1, we might fuse two topics, $t1$ and $t2$, using a function, f, for interpolating time-stamped elements of $t1$ with the single application, $interpolate\ f\ t1\ t2$. We can then apply a binary function, go, to the resulting pairs of values (each including an interpolated element of $t1$ and an element of $t2$), and finally publish the result as a strongly typed ROS topic (here with name `"cmd"`) using the following line of code that defines a functional ROS node,

$$publish\ \texttt{"cmd"}\ (go\ \langle\$\rangle\ interpolate\ f\ t1\ t2)$$

The `roshask`[1] library is compatible with existing clients of the `rostcp` communications protocol. It implements the APIs required of interaction with a ROS master server and with other ROS client nodes, and includes tools to generate Haskell type definitions from standard ROS message type definition files. Integration with the Haskell `cabal` build system allows `roshask` projects to easily draw from the popular `hackage` collection of Haskell libraries.

A. ROS

The ROS software platform [1] is based around architectural principles common to many robotic software frameworks [2]: the abstraction of both hardware and software interfaces to facilitate code reuse and composition.

The elements of ROS that this paper focuses on are *nodes* and *topics*. A node is an independent process that may subscribe to any number of input topics, and publish any number of output topics. A topic is a named channel via which messages of a particular type are multicast from publishers to subscribers.

[1]https://github.com/acowley/roshask

Figure 1 shows an example arrangement in which sensor data gathered by hardware interface nodes are fed to processing nodes whose outputs are sent to a node that integrates those values to produce commands for a motor. The boxes in the diagram represent nodes, while the arrows between boxes correspond to topics. The mapping from diagrams such as this to application specifications is one of the main draws of component oriented design as embodied by ROS.

B. The Limitations of Boxes and Arrows

Unfortunately, the boxes and arrows metaphor tends to stop at the box-implementation level. At this level, the encapsulation provided by the diagrammatic arrows is lost to the underlying series of communication events to be handled by the node. In terms of implementation, a publisher typically invokes an output function to push each new datum to all the subscribers of a particular topic. The subscribers, in turn, must separately handle the events of receiving each new datum on each subscribed topic. In the two most common language bindings for ROS, `roscpp` (C++) and `rospy` (Python), the very act of subscribing to a topic requires providing a callback function that will be invoked whenever a new datum becomes available.

A common scenario is shown in the `Compute3` node in Figure 1: two arrows enter, one leaves. The implication is that the two inputs are both used to generate a single output, making `Compute3` a kind of binary function. Yet separating the event handlers for the input topics puts an onus on the node developer to reclaim the abstraction through some fussy internal plumbing to drive the binary function that produces the node's output.

Simply put, fusing asynchronous inputs is a generic problem with several solutions that should each be implemented once. If a node's behavior is that of a binary function, f, that is to be called every time either input topic produces a value, we write, $f \ \langle\$\rangle \ everyNew \ t1 \ t2$.

Alternatively, if a node's behavior is that of a binary function, g, that is to be called every time both input topics have produced a new value (effectively subsampling the faster topic), we write, $g \ \langle\$\rangle \ bothNew \ t1 \ t2$.

The plumbing that feeds the function belongs in the boxes and arrows metaphor that should persist at every level of abstraction, and this is only possible if we always have a way of referring to a topic in its entirety.

C. Haskell

Haskell[2] is a functional programming language primarily characterized by two distinctive qualities: laziness and purity. As a functional language, Haskell emphasizes the use of functions as first-class values that may be passed to and returned from other functions. Haskell's non-strict, or lazy, evaluation semantics mean that an expression is not evaluated unless it is necessary in producing program output. Purity, indicating that functions are free of side-effects, means that one may always replace two identical function applications appearing in a program with the result of the application.

[2]See the Appendix for a Haskell syntax primer.

Purity helps the programmer reason about a program in an intuitive, equation-driven manner. Of course, not all actions a computer program must perform are pure, mathematical functions. One might consider a "function" like `C`'s $getchar$ to have a type $void \rightarrow Char$, but each time the function is invoked with the same argument, it returns a different value! This $getchar$ action is not a mathematical function, and is instead defined as a value inhabiting the IO monad in Haskell where it is given the type $IO \ Char$ to indicate that it is an input-output action that may be run to produce a value of type $Char$.

Finally, Haskell supports *ad hoc* polymorphism with its typeclass mechanism. A typeclass is a specification of an interface that any type may be made a member of by supplying an implementation of the interface, and is a key mechanism for applying mathematical reasoning to Haskell programs. The distinction between IO values and pure functions, non-strict evaluation, and optimizations due to mathematically rigorous safety properties provide the foundation for a high-level treatment of infinite streams of data that maintains computational efficiency.

II. Related Work

In addition to the robotics frameworks previously cited, there has been some significant work on developing functional approaches to robotics programming [3], [4], [5]. These works all involve Functional Reactive Programming (FRP), a technique that seeks to exploit the algebraic compositionality of time-dependent values and functions on those values. Many implementations of FRP have suffered from performance issues, in part due to difficulty reconciling time-indexed values with finite buffers, and in part due to Haskell's laziness causing unevaluated expressions to grow too large before being evaluated.

A recent application of FRP to robotics, Yampa [6], mitigates these issues by restricting the programming interface to transformations of infinite streams, while the streams themselves remain abstract. The focus on stream transformers is similar to the current presentation, however we endeavour to keep our stream type transparent so that the developer has complete freedom in topic construction. Furthermore, by staying true to ROS design philosophies and implementing full communication protocol compatibility, we do not ask the potential user to forsake any existing software in order to try out a new approach.

The representation and implementation style taken here is greatly informed by Oleg Kiselyov's championing of so-called Iteratees [7]. The driving motivation for this new approach to input iteration is that resource usage should be explicit and carefully managed, in contrast to implicit approaches that rely more heavily on Haskell's lazy evaluation strategy. Existing implementations of Iteratees are not perfectly suited for the proposed application due to our emphasis on infinite streams and sequential input, rather than the finite batched fragments that are the focus of classic Iteratee implementations.

III. Whole-Topic Programming

The notion of *wholemeal programming* [8] revolves around the concept of addressing data structures in their entirety before thinking about the smaller pieces they comprise. This style of programming aims to produce a declarative specification of the *results* of the program, rather than a set of instructions describing how the result should be computed: the *what* as opposed to the *how*. The benefit is that correctness may be more clearly demonstrated through the initial specification, while efficiency may be improved by a series of meaning-preserving transformations of the clear, but possibly inefficient, initial specification.

The key element to this style of programming is that the bulk of the source code should be application-specific. Details such as data structure traversal strategies and generic operations should be encapsulated by standard library functions. The resultant program is then small pieces of interesting application code held together with dabs of higher order glue.

A simple example of the compositionality and clarity afforded by working with whole topics is that of filtering. Suppose we have a topic generated by a sensor, and wish to act whenever the sensor value crosses some threshold. This reactive behavior may be broken into two conceptual parts: *1)* identifying significant sensor readings, and *2)* reacting to those readings. A callback function to be invoked upon each new sensor value might look something like this,

```
void handle_sensor(float val) {
  if(val > threshold) {
    act(val*0.1);
  }
}
```

However, such a design is not fully factored: the threshold-based classification is intermingled with a specific call to action. How the output of the act function is directed to the appropriate output topic is another issue that must be tackled elsewhere. We can instead make the conceptual topic of "all values that pass the threshold" explicit.

Rather than consider each element in isolation, we transform our input topic in its entirety. Fleshing out the example, the input *Topic* is first acquired by subscribing to the `"sense"` topic; it is then filtered against a *threshold* value to produce a topic of values that pass the threshold; the act function is then mapped over the topic of classified inputs to produce a topic of control commands that is finally published under the name `"cmd"`.

$$\textit{subscribe}\ \texttt{"sense"} \gg\!= \\ \textit{publish}\ \texttt{"cmd"} \circ \textit{fmap}\ \textit{act} \circ \textit{filter}\ (>\textit{threshold})$$

This short snippet brings together topic subscription, filtering, transforming, and publishing into a functional ROS node.

A. Operations on Topics

The foundation of fusible functorial mappings and streaming metamorphisms, formally defined in Sections V and VI, provide a rich foundation on which many generic topic transformations may be constructed. We show a few examples here to give the reader a taste of the whole-topic development style. The core `roshask` library provides a collection of functions on *Topic*s that may be directly composed to solve typical topic-oriented problems involving mapping, filtering, prefixes, and suffixes.

These operations may be extended to functions that provide a less rigid mapping from inputs to outputs. For instance,

$$\textit{slidingWindow} :: (\textit{Monad}\ m, \textit{Monoid}\ a) \Rightarrow \\ \quad \textit{Int} \rightarrow \textit{Topic}\ m\ a \rightarrow \textit{Topic}\ m\ a$$

passes a sliding window of the given integer size along an input *Topic* producing an output *Topic* whose values are the monoidal composition of all the values covered by the window.

A more efficient version, called $slidingWindowG$, is defined for all algebraic groups. Suppose we have a topic carrying floating point values, and we want to consider the moving average over 10 consecutive values by relying on *Float* being an additive group (captured by the $AdditiveGroup$ typeclass).

$$\textit{avg} :: \textit{Monad}\ m \Rightarrow \textit{Topic}\ m\ \textit{Float} \rightarrow \textit{Topic}\ m\ \textit{Float} \\ \textit{avg} = \textit{fmap}\ (*0.1) \circ \textit{slidingWindowG}\ 10$$

This function first transforms the original topic into a topic carrying sums of 10 consecutive values by subtracting an old element from the running sum for each new element added, then normalizes those sums with a multiplication.

B. Topic Synchronization

The fusion of multiple asynchronous topics is a motivating capability for `roshask`. The core library provides several functions on two topics that return a new topic consisting of:

- An interleaving of two topics formed by injecting items from either topic into the $Either$ data type.
- Pairs of values that are produced every time either input topic produces a value. The pair consists of the newly produced value from one topic, and the most recently produced value from the other topic.
- Pairs of values that are produced every time a new value is available from both input topics. If one topic produces values faster than the other, some of its values will be dropped.
- Values from both topics transparently merged. If the two input topics produce values of the same type, they may be merged into a single topic.

A more complex merging of two topics was referred to in the introduction: timing-based interpolation. Many ROS message types include a header field containing a time stamp. Using these time stamps, we may interpolate one topic so that we have an approximate partner for each value from a faster-moving topic. Consider a sensor outputting scans at 20Hz that are to be paired with a position estimate that is updated at 5Hz. If we have a function, $interp$, that linearly interpolates two robot poses, we can "pose stamp" each sensor scan.

$interpolate\ interp\ poses\ scans$

The type of the resulting topic depends on the input topics; an application to a topic of *PoseStamped* messages and a topic of *LaserScan* messages (from the `geometry_msgs` and `sensor_msgs` ROS stacks) has the type,

$$Topic\ IO\ (PoseStamped, LaserScan)$$

This topic comprises pairs that include an interpolated *PoseStamped* for each *LaserScan*. The key consideration is that *interpolate* is constrained to operate on types that are instances of the *HasHeader* typeclass that `roshask` defines for ROS message types that have a header field. This gives *interpolate* enough information to identify pairs of consecutive elements from the first topic that temporally bracket elements from the second topic, apply a user-supplied linear interpolation function of type $a \to a \to Double \to a$ to the bracketing pair, and produce a result pair.

IV. ROS Nodes

With so much functionality pushed into topics, the responsibility of a node is limited to initialization and the naming of input and output topics. A useful ROS tutorial project involves creating a publisher (*Talker*) node that sends text messages at 1Hz to a subscriber (*Listener*) node that prints those messages. In `roshask`, the Talker side of this system is broken into two pieces: the *Topic* of string messages, and the *Node* that names and regulates the output topic.

The messages generated by this *Node* consist of a greeting and the current time packaged up in a ROS String message type. The module defining this message is imported qualified with an "*S*." prefix to avoid name collisions, allowing us to define the output topic as,

$$\begin{aligned} &sayHello :: Topic\ IO\ S.String \\ &sayHello = repeatM\ (fmap\ mkMsg\ getTime) \\ &\quad \textbf{where}\ mkMsg = S.String \circ (\texttt{"Hi "} +\!\!+) \circ show \end{aligned}$$

This *Topic* is an infinite stream of ROS String messages, each produced by querying the system for the current time with the *getTime* action, formatting the time as a Haskell *String* using the *show* function, prepending the string `"Hi "`, and packaging the Haskell *String* up into a ROS String message using the *S.String* data constructor. This definition is a minimal statement of the data carried by the output topic.

The second piece of the Talker is the Node definition, which here has two responsibilities: name the output topic, and limit its production rate to 1Hz.

$$\begin{aligned} &tn :: Node\ () \\ &tn = advertise\ \texttt{"chatter"}\ (topicRate\ 1\ sayHello) \end{aligned}$$

Note that the *advertise* expression produces a value of type *Node* (). The *Node* type used by `roshask` is a stack of monad transformers outside the scope of this paper. In brief, a *Node* is a monad that has read-only configuration data (*e.g.* from arguments like topic remappings passed to a ROS executable) and dynamic state tracking publication and subscription information on top of the usual *IO* monad that allows the programmer to perform any initialization necessary to get a node started.

In order to give the node a name, `"talker"`, to register with the ROS master server, and start things running, we provide one more definition to provide the executable an entry point.

$$\begin{aligned} &main :: IO\ () \\ &main = runNode\ \texttt{"talker"}\ tn \end{aligned}$$

The Listener side of the system is symmetric, and also imports the module defining the ROS String message type qualified with an "*S*." prefix. As we wish to simply take an action for every input message – print it out to the screen – a function from String messages to *IO* actions is called for,

$$\begin{aligned} &showMsg :: S.String \to IO\ () \\ &showMsg = putStrLn \circ (\texttt{"Msg: "} +\!\!+) \circ S.\_data \end{aligned}$$

This function projects the *_data* field of the String message record type, prepends it with the string `"Msg: "`, and prints the resultant Haskell *String* to standard output.

Finally, we define and run the Listener node,

$$\begin{aligned} &main = runNode\ \texttt{"listener"}\ \$ \\ &\quad runHandler\ showMsg =\!\!\ll subscribe\ \texttt{"chatter"} \end{aligned}$$

This defines a *Node* that *subscribe*s to a named topic, producing a value of type, *Node* (*Topic IO S.String*). The *Topic* is piped straight into *runHandler showMsg* that applies the *showMsg* function to each input message and runs the resulting *IO* action. The constructed *Node* is, as usual, run with the *runNode* function (the $ operator is simply a stand-in for function application whose low binding precedence permits the elision of parentheses).

A. Composition of Nodes

The Talker and Listener nodes are compiled into standalone executables that register with the ROS master server, then exchange messages until one or both are shut down. Since node definitions are compositional, monadic values, it is a simple matter to glue multiple node specifications together in order to run the composite node in a single process. When a *subscribe* action is taken in the *Node* monad, a check is performed to see if a publisher of the selected topic has previously been registered in this *Node*. If so, the publisher's output *Topic* is fed directly to the subscriber with no intermediation by the usual `rostcp` communication protocol.

This ability to flatten *Node* structure does not require any changes to *Node* or *Topic* definitions. The same *Node* may be run as a standalone process with *runNode*, or imported into another Haskell module and combined with other *Node* values to produce a composite *Node* that will exploit serialization-free *Topic*-based data interchange.

Table I shows a performance comparison of a worst-case scenario for message copying overhead: a high-resolution video producer feeds uncompressed video frames (1024x1024 resolution at 60Hz) to a simple analysis node.

	Total CPU (%)	Total RAM (MB)
Separate Processes	8.3	23.7
Single Process	4	9.6

TABLE I
VIDEO PRODUCER AND CONSUMER (CPU: INTEL CORE I5)

Serialization through the TCP stack has a dramatic effect on performance and resource usage that is virtually eliminated by arranging for the two nodes to run in the same process. The statistics shown in the table are drawn from operating system-level performance monitors, but we can gain more insight into exactly what the executables are doing.

The Glasgow Haskell Compiler (GHC) runtime system (RTS) tracks several metrics on the runtime behavior of the executables, and reports that the single-process variant uses a maximum of just over 2MB of RAM for application data[3]. This lines up with the expectation that the program requires at least two video frames: one the producer is writing into, and one the analyzer is processing. This efficient use of memory is purely thanks to the compiler; the ROS nodes make no accommodations for memory management. The RTS further reports that the garbage collector used 8.5% of the total CPU time of the program. This overhead is in exchange for totally asynchronous production and consumption fitting into a near optimal amount of memory.

In comparison, running the video producer in a separate process uses a bit over 2MB of RAM for application data, while the standalone consumer process uses a bit over 3MB of RAM. The extra memory usage is due to serialization and communication coordination on each end effectively introducing an extra copy of each video frame in each process. The duplicated overhead of asynchrony is removed when the two nodes are run in the same process.

The two ROS nodes in this example are defined in separate modules that may be compiled and linked as standalone executables in addition to the composite executable. The entire source code of the module defining and running the composite node is,

$$
\begin{aligned}
&\textbf{module}\ \mathit{Main}\ (\mathit{main})\ \textbf{where}\\
&\textbf{import}\ \mathit{Ros.Node}\\
&\textbf{import}\ \mathit{VidProducer}\ (\mathit{producer})\\
&\textbf{import}\ \mathit{VidConsumer}\ (\mathit{consumer})\\
&\mathit{main} = \mathit{runNode}\ \texttt{"NodeCompose"}\ \$\\
&\quad \mathit{producer} \gg \mathit{consumer}
\end{aligned}
$$

The $\mathit{VidProducer}$ and $\mathit{VidConsumer}$ modules define Nodes that publish and subscribe to a particular Topic. By importing them into one module, and composing them with the $\gg$ operator on Monads, we eliminate serialization overhead without impacting the modularity of the design. No change to node definitions or topic usage is required to use nodes in standalone processes or as part of a larger, single-process node that eliminates serialization overhead.

[3]This is what GHC calls "maximum residency", in contrast to the total amount of RAM allocated from the operating system for use by the garbage collector. This number represents how much data the collector sees when it runs, and reflects how much live data the program is working with.

V. FIRST-CLASS TOPICS

In order to provide libraries of topic-level combinators, we require a useful representation for ROS topics. This representation must be parametric over the underlying message type, must not lead to surprising memory leaks or performance degradation, and must present enough structure that it may be concisely manipulated by familiar algebraic power tools.

The type we want is not an inductive list, for it is always infinite, nor is it simply a coinductive stream[4] of pure values. The values produced by a topic might be gathered from some hardware device or received over a network connection; each discrete step taken by a topic may be an impure computation that must be reflected by some context. Putting these elements together, we say that a topic is a step function in some computational context that yields a pair of a value and the rest of the topic.

The type we use for ROS topics is (we abbreviate Topic as T here for space considerations),

$$\textbf{newtype}\ T\ m\ a = T\ \{\mathit{unT} :: m\ (a, T\ m\ a)\}$$

This defines a binary type constructor, T, with a single unary data constructor, also written, T, that takes a single argument of type, $m\ (a, T\ m\ a)$. This argument is a pair of a value of type a and a topic of type, $T\ m\ a$ (*i.e.* the rest of the topic), all wrapped by some other type constructor represented by m. We give a name, unT, to the function that strips off a T data constructor to get access to the $m\ (a, T\ m\ a)$ payload.

A. Topics are Functors

The central feature we wish to verify about a $T\ \ m$ structure (polymorphic in the message type) is that it is a *functor*. A functor is an algebraic structure, sometimes referred to as a structure-preserving map, that provides a mechanism of lifting functions into the functor structure. In Haskell, this mapping behavior is captured by the $\mathit{Functor}$ typeclass that identifies a single function,

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ t\ \textbf{where}\\
&\quad \mathit{map} :: (a \to b) \to t\ a \to t\ b
\end{aligned}
$$

[*Note:* In Haskell, the function map shown above is called fmap. We write map in this section as there is no ambiguity with Haskell's map function (which is fmap specialized to lists), and we again seek to keep notation compact.]

The type of this function is not enough to establish a correspondence between Haskell's $\mathit{Functor}$ and a mathematical functor, so one must also check that the so-called functor laws are satisfied for a candidate structure.

[4]Recall that a stream data type has a single constructor that pairs an element of the stream with the rest of the stream.

$$map\ id \equiv id \quad (1)$$
$$map\ f \circ map\ g \equiv map\ (f \circ g) \quad (2)$$

The second equation is the more interesting: it tells us that if we wish to apply two functions to the elements of a functorial structure, we do not need to traverse the structure more than once. Instead, we can simply compose the functions and apply that composition in a single traversal of the structure. A compiler may perform this optimization for us, allowing us to maintain modular definitions while still gaining the benefit of this traversal fusion. It should be noted that this optimization is absolutely critical for our *Topic* types whose inhabiting values are infinite.

In order to prove that our structure satisfies this law, we will require that the first type parameter of T, what we have called m, is a functor itself. With that, here is the definition of map for $T\ m$,

$$\textbf{instance}\ \mathit{Functor}\ m \Rightarrow \mathit{Functor}\ (T\ m)\ \textbf{where}$$
$$map\ f\ (T\ t) = T\ (map\ (f \otimes map\ f)\ t)$$

We can clarify this definition by annotating each usage of the map function with the type it is used at, with the proviso that we write map_T to indicate what should properly be written $map_{(T\ m)}$.

$$map_T\ f\ (T\ t) = T\ (map_m\ (f \otimes map_T\ f)\ t)$$

The $\otimes$ function we are using is actually the $***$ function from the Haskell standard library's *Arrow* type class. For expository purposes, we consider an approximation of that library function specialized to the type of ordinary functions.

$$(\otimes) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d)$$
$$f \otimes g = \lambda(x, y) \to (f\ x, g\ y)$$

B. Proving Functoriality

We reason about our types and functions using equational reasoning [9], wherein a chain of equalities relates two expressions with the justification for each equality written in braces between each step. Haskell lends itself to this approach thanks to laziness and referential transparency [10] that permit one to substitute equals for equals without worrying about hidden side effects or unwarranted evaluation.

Since a topic carries an infinite amount of data, the proof must be by coinduction. The topics on either side of the equation are observed, or destructed, in the only possible way to extract a single pair value from each whose first projections are equal. The second projections of the pairs, representing the rest of each respective topic, correspond exactly to the bisimilarity being demonstrated. The corecursive equality is said to be guarded by the pair constructor which provides the structure for a single-step observation of each topic that may be repeated to witness an arbitrary-length finite equality.

Theorem *((T m) is a functor):*

$$map_T\ f \circ map_T\ g \equiv map_T\ (f \circ g)$$

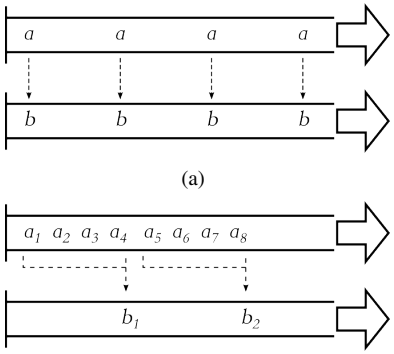

Fig. 2. (a) Mapping a function with type $a \to b$ over a topic with type $T\ m\ a$ resulting in a topic with type $T\ m\ b$. Topics are visualized as ordered streams of values starting whenever a program begins observing them and continuing forever. (b) A metamorphism over a topic with type $T\ m\ a$ resulting in a topic with type $T\ m\ b$. In this case, four elements of type a are consumed to produce each element of type b.

Proof:

$$\begin{aligned}
&(map_T\ f \circ map_T\ g)\ (T\ t)\\
\equiv\ &\{\ \text{definition of function composition}\ \}\\
&map_T\ f\ (map_T\ g\ (T\ t))\\
\equiv\ &\{\ \text{definition of } map_T\ \}\\
&map_T\ f\ (T\ (map_m\ (g \otimes map_T\ g)\ t))\\
\equiv\ &\{\ \text{definition of } map_T\ \}\\
&T\ (map_m\ (f \otimes map_T\ f)\ (map_m\ (g \otimes map_T\ g)\ t))\\
\equiv\ &\{\ \text{definition of function composition}\ \}\\
&T\ ((map_m\ (f \otimes map_T\ f) \circ map_m\ (g \otimes map_T\ g))\ t)\\
\equiv\ &\{\ \text{functor law for } m\ \}\\
&T\ (map_m\ ((f \otimes map_T\ f) \circ (g \otimes map_T\ g))\ t)\\
\equiv\ &\{\ \text{functor law for the product bifunctor}\ \}\\
&T\ (map_m\ ((f \circ g) \otimes (map_T\ f \circ map_T\ g))\ t)\\
\equiv\ &\{\ \text{coinduction hypothesis}\ \}\\
&T\ (map_m\ ((f \circ g) \otimes (map_T\ (f \circ g)))\ t)\\
\equiv\ &\{\ \text{definition of } map_T\ \}\\
&map_T\ (f \circ g)\ (T\ t)
\end{aligned}$$

□

VI. Topic Metamorphisms

The functorial structure of topics allows us to map functions over the values carried by the topics, but this implies that every input message leads to exactly one output message. Figure 2(a) visualizes the effect of mapping a function over a topic; each element in the result topic corresponds to a specific element from the input topic.

In order to gain flexibility in how input messages may be used to produce output messages, we must generalize the ways in which a topic may be consumed. The consumption, or tearing down, of data structures is universally captured by the concept of a *fold*. The reverse direction, data production, is implemented by *unfold* functions that thread a seed value through a recursive construction procedure.

The usual arrangement of these functions in Haskell programming is the composition of a fold with an unfold,

$fold \circ unfold$. A naïve reading of this composition suggests that the $unfold$ produces some large structure, which is then torn down by the fold. Clever compilers can, however, deforest such a computation and eliminate the intermediate data structure. Gibbons [11] identifies the reverse composition, $unfold \circ fold$, as transforming one data structure into another by tearing down the first to produce a seed that is used to germinate the second. This composition also lends itself to an optimization in that its output may, under a *streaming condition* [11], be incrementally consumed, thus obviating the need for the initial fold to fully traverse the input structure before useful output is produced.

The precise way we apply this so-called *metamorphism* [5] to topics is with a higher order function, $metamorph$, that takes a function whose output is an optional output value along with a continuation function for handling subsequent inputs. The continuation function may be used to thread state through the deconstruction of one topic, while the optional output values are used to construct a new topic. An example of such a transformation is shown in Figure 2(b), demonstrating the contraction mapping behavior metamorphisms offer over standard functorial mapping.

Gibbons's streaming condition is a statement about causality: we may release a component of the output of the unfold when we know that its value does not depend on subsequent inputs. This ties in to the very real issue of resource management when trying to address infinite structures such as topics. Even though a topic is infinite, we may safely compute with it as long as we only ever consider a finite portion of it.

VII. Conclusion

Working with whole topics encourages a top-down approach to component design. By thinking about properties that should hold of all the outputs of a node, or of how all outputs are related to inputs, one is able to more easily identify the essence of a node. At every stage, we wish to support design plans that begin, "If I had a topic carrying *this* kind of value...." That approach to design is precisely the right attitude as it is true and faithful to the tenets of modular, stream-oriented programming. The `roshask` tools allow the ROS programmer to practice holistic design while retaining high performance and access to the rich libraries of device interfaces and controllers developers have written to ROS interfaces.

ACKNOWLEDGMENTS

The authors thank Gershom Bazerman for his key contributions at the genesis of this project.

Appendix: Notation Primer

Core Haskell notational conventions are shown in Table II. Caution is warranted as there is some reuse of names and syntax between the type and value levels, but the context in which the syntax is found is always unambiguous.

[5] $folds$ and $unfolds$ are sometimes referred to as *catamorphisms* and *anamorphisms*, while the composition $fold \circ unfold$ is referred to as a $hylomorphism$ [12]. It is that naming scheme that provoked the coining of the name *metamorphism*.

Notation	Interpretation
x, y, z	variables representing values
a, b, c	type variables
S, T	type constructors and data constructors
$x :: t$	x has type t
$C\ x \Rightarrow x$	type x is an instance of type class C
$a \rightarrow b$	The type of a function from a to b
$\lambda x \rightarrow f$	A function that binds argument x in its body, f
$f\ x$	The application of function f to x
$f \circ g$	The composition of functions f and g. $(f \circ g)\ x \equiv f\ (g\ x)$
(x, y)	A pair, or product, of values x and y
(a, b)	The type of pairs whose first component is of type a and whose second component is of type b

TABLE II
Notation Reference

References

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *Proceedings of the Open-Source Software workshop at the International Conference on Robotics and Automation (ICRA)*, 2009.

[2] A. Harris and J. Conrad, "Survey of popular robotics simulators, frameworks, and toolkits," in *Southeastcon, 2011 Proceedings of IEEE*, march 2011, pp. 243 –249.

[3] G. D. Hager and J. Peterson, "Frob: A transformational approach to the design of robot software," in *In Robotics Research: The Ninth International Symposium*. Springer Verlag, 1999, pp. 257–264.

[4] J. Peterson, P. Hudak, and C. Elliott, "Lambda in Motion: Controlling Robots with Haskell," *Lecture Notes in Computer Science*, vol. 1551, pp. 91–105, 1999.

[5] J. Peterson and G. Hager, "Monadic Robotics," in *Domain-Specific Languages*, 1999, pp. 95–108.

[6] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Summer School on Advanced Functional Programming 2002, Oxford University*, ser. Lecture Notes in Computer Science, vol. 2638. Springer-Verlag, 2003, pp. 159–187.

[7] O. Kiselyov, "Incremental multi-level input processing with left-fold enumerator," in *Developer Tracks on Functional Programming (DEFUN)*. ACM SIGPLAN, 2008. [Online]. Available: http://okmij.org/ftp/Haskell/Iteratee/DEFUN08-talk-notes.pdf

[8] R. Bird, "Functional pearl: A program to solve sudoku," *Journal of Functional Programming*, vol. 16, pp. 671–679, November 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1180085.1180089

[9] R. Bird and O. de Moor, *Algebra of programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[10] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, pp. 98–107, 1984.

[11] J. Gibbons, "Metamorphisms: Streaming representation-changers," *Science of Computer Programming*, vol. 65, pp. 108–139, 2007. [Online]. Available: http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/metamorphisms-scp.pdf

[12] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 124–144. [Online]. Available: http://portal.acm.org/citation.cfm?id=127960.128035