Functional Reactive Programming with Effects, a more permissive approach

Frédéric Dabrowski, Jordan Ischard

FUNCTIONAL REACTIVE PROGRAMMING WITH EFFECTS, A MORE PERMISSIVE APPROACH

A PREPRINT

Frédéric Dabrowski[1] and Jordan Ischard[1]

[1]Univ.Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France

frederic.dabrowski@univ-orleans.fr jordan.ischard@univ-orleans.fr

ABSTRACT

We introduce a functional reactive programming language that extends WORMHOLES, an enhancement of YAMPA with support for effects. Our proposal relaxes the constraint in WORMHOLES that restricts all resources to single-use. Resources are categorized into two kinds: input/output resources and internal resources. Input/output resources model interactions with the environment and follow constraints similar to those in WORMHOLES. Internal resources, on the other hand, enable communication between program components and can be used multiple times. We demonstrate that programs written in our language can be translated into equivalent effect-free YAMPA programs, ensuring that our approach remains compatible with existing functional reactive paradigms.

1 Introduction

Reactive systems, which maintain continuous interaction with their environment, are essential for modern applications such as user interfaces, simulations, and control systems, as highlighted by Harel and Pnueli who first characterized these systems [11]. One particularly influential line of research has led to several Haskell libraries, originating from FRAN (Functional Reactive ANimation), as proposed by Elliott and Hudak [8]. These libraries are often categorized under Functional Reactive Programming (FRP), though the term FRP can also encompass a broader family of languages. They provide a declarative approach to programming reactive systems, where computations are expressed as functions over time-varying values, known as signals or streams.

FRAN was designed to simplify the development of animation and simulation applications in Haskell. Over time, various proposals have optimized the execution model [9], enriched the language [28, 26, 20], and addressed limitations [2, 24, 12]. One of FRAN's most notable drawbacks was its inherent space leak, caused by the system retaining past signal values unnecessarily. The FRPNOW library, designed by Ploeg and Claessen [24], addresses this issue by imposing restrictions on the use of signals, allowing the system to discard past values. The YAMPA library, designed by Hudak *et al.* [12] takes a more radical approach by not treating signals as first-class citizens. In this approach, programs are arrows, a generalization of monads introduced by Hughes [13]. Intuitively, arrows are to morphisms what monads are to objects. They denote stateful transformers, that generate an output and an updated transformer from each input. The latter is then applied to the next input, continuing the process iteratively. The arrow structure of programs allows them to be composed to build more complex systems. This approach aligns YAMPA with synchronous languages such as ESTEREL [4], SIGNAL [3], LUSTRE [10] and LUCIDSYNCHRONE [18], which operate based on a logical notion of time. At each step (or tick of the logical clock), the system produces an output depending on the current input, thus capturing the essence of length-preserving synchronous functions as defined by Caspi and Pouzet [7]. The declarative nature of YAMPA aligns it more closely with data-flow-oriented programming languages such as SIGNAL and LUSTRE, as opposed to control-flow-oriented programming languages such as ESTEREL. Libraries for general-purpose programming languages have also been proposed for the latter paradigm, including REACTIVEC [5], SL [6], and REACTIVEML [15].

Despite YAMPA's advantages, managing I/O operations remains challenging due to the lack of support for monadic programming in the language. An early attempt to address this limitation was proposed by Winograd-Cort and Hu-

dak [27, 26], who introduced resource signal functions (RSFs). RSFs facilitate imperative-style resource access within a functional framework. Resources can be either global, representing inputs and outputs, or local, representing local resources. In a pure functional setting, this approach has since been superseded by Monadic Stream Functions (MSFs), introduced by Perez *et al.* [17]. Roughly speaking, MSFs extend the Arrow framework of YAMPA by embedding a monadic structure into the type of signal functions. This concept has been implemented in the DUNAI and FRPBEARRIVER libraries, the latter being a refactored version of YAMPA, built on top of DUNAI. Despite these advancements, we believe that WORMHOLES provides a solid foundation for adapting the model to an impure language such as OCaml, where monads are less naturally integrated. In this context, resources could be represented using mutable references. This originally served as our primary motivation for the work presented in this paper.

WORMHOLES provides a simple and elegant way to manage resources. However, the original model imposes strict access constraints, allowing resources to be accessed only once. For inputs and outputs, this restriction aligns well with the synchronous hypothesis, which states that each computational step should appear as an atomic operation to the environment. In synchronous language terminology, such computations are said to take zero time. However, for local resources, this restriction may be overly rigid.

In this paper, we introduce MOLHOLES, a language that extends the WORMHOLES approach by relaxing resource usage constraints. MOLHOLES adopts a three-resource paradigm consisting of inputs, outputs, and internal resources. It drops the single-access restriction on local resources, allowing multiple reads and writes. However, input resources must be read exactly once, and output resources must be written exactly once. The latter property ensures that programs are productive. We formalize the semantics of MOLHOLES and present a trivial static analysis that ensures correct resource usage. We also establish that well-typed programs can be transformed into equivalent programs with two key properties. The first property states that all resources, including internals, resources are read and written at most once. The second property states that all read operations occur at the beginning of computation, while all write operations occur at the end. This aligns with the synchronous hypothesis. Finally, we demonstrate that this transformation enables converting our programs into resource-free Yampa programs while preserving their semantics. To achieve this, we formalize YAMPA's semantics and establish the correctness of our transformation using equational reasoning and bisimulation techniques. As an introductory example, we demonstrate how bisimulation can be used to establish two well-known properties: that YAMPA's semantics domain indeed form an Arrow and that YAMPA programs admit a kind of normal form. Notably, the form of programs generated by the first transformation can also be considered a normal forms in MOLHOLES, while the second transformation maps this normal form to a YAMPA program in normal form. Except for resource accesses, which require specific equalities, the transformations rely on standard equational theories of categories, functors, monads, and arrows.

In section 2, we provide a brief overview of YAMPA and WORMHOLES. In section 3, we introduce some basic equational theory of categories, functors, monads, and arrows. We also recall the definition of coalgebras and bisimulation. In section 4, we formalize a core subset of YAMPA and employ bisimulation techniques to demonstrate that the semantics satisfies Arrows equations and that programs admit a normal form. In section 5, we introduce MOLHOLES, formalize its semantics, and prove the correctness of its type system. In section 6, we present the two transformations mentioned above. Finally, in section 7, we summarize our contributions and outline directions for future work.

2 Overview

2.1 YAMPA

The YAMPA library offers a domain-specific language, built on top of HASKELL, designed for programming reactive systems. It offers a rich set of transformers to define functions which transform input streams into output streams. These transformers, commonly referred to as signal functions, are represented as instances of the `Arrow` type class. In YAMPA, signal functions have the type $sf\ A\ B$, where A and B are HASKELL types representing the types of values carried by input and output signals, respectively. The primary signal functions include arr, $first$, $comp$, and $loop$, with $comp$ typically written in infix notation as $\cdot >>> \cdot$. Their behaviors are outlined below, where f denotes a HASKELL function of type $A \to B$ and sf, sf_1 and sf_2 represent signal functions. A graphical representation of these signal functions is provided in fig. 1 and a formal semantics will be presented in section 4.

- arr has the type $(A \to B) \to sf\ A\ B$. It transforms a HASKELL function into a signal function. For example, $arr\ (*2)$ represents a signal function that doubles a numeric input.
- $first$ has the type $sf\ A\ B \to sf\ (A \times C)\ (B \times C)$. Given a signal function sf, $first\ sf$ processes the first component of the input signal with sf while leaving the second component unchanged.
- $comp$ has the type $sf\ A\ B \times sf\ B\ C \to sf\ A\ C$. It composes two signal functions sequentially.

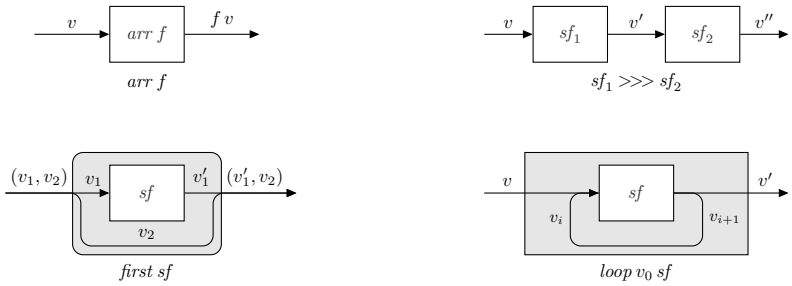
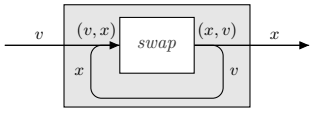


Figure 1: Graphical representation of primary signal functions

- *loop* has the type $C \to sf\ (A \times C)\ (B \times C) \to sf\ A\ B$. Given a value v and a signal function sf, $loop\ v\ sf$ is a signal function with internal feedback, enabling stateful computations.

The value v in $loop\ v\ sf$ represents the initial state of the computation. In fig. 1, the value v_i represents the dynamically evolving internal state. For example $delay\ x = loop\ x\ (arr\ swap)$, where $swap = \lambda(x, y).(y, x)$, represents a signal function that takes as input a signal carrying values of type A and produces an output signal with the same values, delayed by one step and starting with the initial value x.



A signal function of type $sf\ A\ B$ takes an input of type A and produces both an output of type B and an updated version of itself. When processing an input stream, the signal function is applied to the stream's head to produce the current output. The update signal function is then applied to the tail of the stream, allowing the state to evolve step-by-step.

2.2 WORMHOLES

The WORMHOLES library extends YAMPA by introducing new constructs for controlled side effects. At the core of this extension is the concept of resources, which are identifiers attached to side effects. Each resource can be accessed at most once. Signal functions, or resource signal functions, have types of the form $A \overset{R}{\rightsquigarrow} B$, where A and B are the types of input and output values, respectively. The annotation R denotes the set of resources involved in the computation and is used by the type system to ensure correct resource usage. To access resources, WORMHOLES introduces a construct, named rsf, which creates a signal function $rsf[r]$ from a resource identifier r. This signal function takes an input value to be consumed by the environment and returns an output value produced by the environment. A resource r has a type $\langle A, B\rangle$, where A and B specify the types of inputs and outputs, respectively. Given such a resource r, the signal function $rsf[r]$ has type $A \overset{\{r\}}{\rightsquigarrow} B$.

Resources are classified as either unbound or bound. Unbound resources represent interactions with the real world, while bound resources represent communication channels between components of the program. To manage bound resources, WORMHOLES introduces the construct

$$wormhole[r_{get}; r_{set}](t_0; t)$$

which binds the resources r_{get} and r_{set} within the term t. The term t_0 represents the initial value of the resource r_{get} while r_{set} is initialized with `tt`. The first resource is used for reading a value from the channel and must have a type of the form $\langle Unit, A\rangle$, while the second resource is used for writing a value to the channel an must have a type of the form $\langle A, Unit\rangle$. The type of such a term matches that of t, except that r_{get} and r_{set} are removed. Regardless of whether resources are bound or unbound, they are single-use, meaning they can be accessed at most once.

In addition to the rsf and $wormhole$ constructs, WORMHOLES provides the signal functions arr, $first$ and $comp$ which are similar to their YAMPA counterparts. Unlike YAMPA, WORMHOLES does not include a $loop$ function. However, as demonstrated by Winograd-Cort and Hudak, it can be emulated. Specifically, a term of the form $loop\ v_0\ t$

can be encoded as

$$\begin{aligned} \mathit{wormhole}[r_{get}, r_{set}](v_0; \mathit{arr}\,(\lambda x.(\mathtt{tt}, x)) >\!\!>\!\!> \mathit{first}\,(\mathit{rsf}[r_{get}]) >\!\!>\!\!> \\ \mathit{arr}\,(\lambda(x,y).(y,x)) >\!\!>\!\!> t >\!\!>\!\!> \mathit{arr}\,(\lambda(x,y).(y,x)) >\!\!>\!\!> \\ \mathit{first}\,(\mathit{rsf}[r_{set}]) >\!\!>\!\!> \mathit{arr}\,\mathit{snd}) \end{aligned}$$

where r_{get} and r_{set} are fresh resources, v_0 is the initial value of the loop, and t is its body.

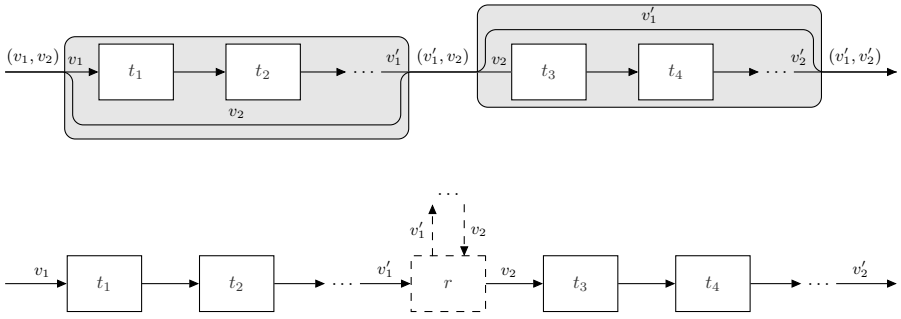


Figure 2: Equivalent programs in YAMPA (top) and in WORMHOLES (bottom)

The use of resources in WORMHOLES enhances modularity. Consider a program that performs two tasks sequentially: the first task sends a value to the environment, while the second task receives a value from the environment. In YAMPA, both the input stream and the output stream must carry a pair, as illustrated in the top schema of fig. 2. In contrast, an equivalent WORMHOLES program requires only the first component, while defining a resource r to represent the interaction "I provide my first result in exchange for the expression needed for my second task", as depicted in fig. 2.

To enforce an affine use of resources at the semantics level, WORMHOLES introduces a memory cell for each resource, see fig. 3 where the resource r is associated with the memory cell c. Each memory cell carries a tag that indicates whether its associated resource has been used. The cell consists of tow sub-cells, one for the input and one for the output. However, a cell cannot hold both pieces of information simultaneously. The tag is inferred from the shape of the cell. If the left sub-cell contains an element while the right sub-cell contains (-), the cell is accessible. At the beginning of each computation step, all cells are accessible. When a resource is accesses through the rsf construct, the state of cell updates accordingly. The type system of WORMHOLES ensures correct resource usage by tracking their accesses.

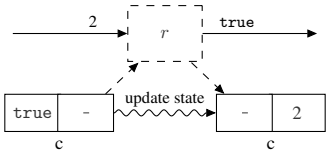


Figure 3: Graphical representation a $rsf[r]$ term computation.

The problem outlined in this section naturally boils down to the usual challenges of handling effects in a purely functional language. Monads provide an effective solution, and as mentioned in the introduction, YAMPA's model has been adapted to support them. As we also noted, it is with the objective of designing a library for an impure functional language that we build upon the WORMHOLES's model.

3 Semantic Domains

In sections section 4 and section 5, we formalize the semantics of YAMPA and MOLHOLES, and in section 6, we establish the correctness of various program transformations. These transformations occur both within each language and between the two. Both languages are built upon a common abstract host language, axiomatized at the semantics level by a collection of typed values and typed functions forming a cartesian category $\mathcal{C}_{\text{hst}}$. The cartesian requirement ensures that product types and functions over product types are available in the host language, which is essential for our transformations.

Programs in YAMPA and MOLHOLES represent length-preserving synchronous functions. Their semantics is defined by iteratively applying a step function. The semantic domains of these step functions can be modeled as terminal coalgebras of a family of functors. This family, indexed by types of the host language, is defined as follows:

$$\begin{array}{lcl} F_{A,B}\,X & = & A \to (B \times X) \\ F_{A,B} & : & (X \to Y) \to (F_{A,B}\,X \to F_{A,B}\,Y) \\ F_{A,B}\,f & = & \lambda step\,a.\texttt{let}\ (b,x) = step\,a\ \texttt{in}\ (b, f\,x) \end{array}$$

Given two types A and B and a coalgebras (X, f) for $F_{A,B}$, X represents a type of transformers and f is a morphism mapping a transformer to a step function. Intuitively, this function applies the transformer to an input value of type A, producing an output value of type B along with a new transformer. In the case of MOLHOLES, the coalgebra incorporates a state monad to express the semantics of effects. A transformation from MOLHOLES to YAMPA is then expressed as a coalgebra morphism between their respective semantics domains. In both cases, the carrier of the coalgebra forms an arrow, as defined by Hughes [13]. The correctness of the transformation is established by applying bisimulation proof techniques between coalgebras of $F_{A,B}$, we obtain a powerful method for establishing behavioral equivalence (bisimilarity) both within a single language and between the two languages. Intuitively, bisimilarity between two transformers expresses the idea that they exhibit the same behavior. In the context of YAMPA and MOLHOLES, this means that, for any given input, the transformers produce the same output and that their updated transformers remain bisimilar. Bisimilarity is required because our semantics domain consists of corecursive functions. Proving equivalence between corecursive functions requires coinductive reasoning techniques, which do not apply to Leibniz equality.

In the remainder of this section, we provide a brief overview of the equational theories of categories, functors, monads, and arrows. Additionally, we define bisimulation between elements of two coalgebras of the functor $F_{A,B}$. This section, which may be skimmed on a first reading, presents only the essential material needed to justify the transformations applied later in the paper. For an in-depth introduction to these concepts, we refer the reader to the works of Spivak [21], Jacobs [14], and Sangiorgi [19]. Our presentation of monads and arrows follows the functional programming style, which differs slightly from the category-theoretic approach. For a comprehensive introduction to monads in the context of functional programming, we recommend Wadler's paper [25], while Hughes's paper [13] introduces arrows similarly. A comprehensive study of Monads can be found in Moggi's seminal paper [16]. An in-depth study of the relationship between arrows and Freyd categories can be found in Atkey's paper [1].

3.1 Categories

A category $\mathcal{C}$ consists of a collection of objects, which we denote by $\mathcal{C}$, and a collection of morphisms for each pair of objects A and B, which we denote by $\mathcal{C}\,A\,B$. Additionally, a category $\mathcal{C}$ includes:

- a family of identity morphisms id_A for each object A
- a family of composition operators $\circ_{ABC} : \mathcal{C}\,A\,B \times \mathcal{C}\,B\,C \to \mathcal{C}\,A\,C$ for each triple of objects A, B, and C.

These morphisms and operators must satisfy the following laws:

$$f \circ id = f \tag{1a}$$
$$id \circ f = f \tag{1b}$$
$$(f \circ g) \circ h = f \circ (g \circ h) \tag{1c}$$

An object A is terminal if for all objects B, there exists a unique morphism $f : \mathcal{C}\,B\,A$. Terminal objects are unique up to isomorphism. A cartesian category is a category with a terminal object and binary products for all pairs of objects. For a pair of objects A and B, the product of A and B is an object $A \times B$ with two projection morphisms $\pi_1 : \mathcal{C}\,A \times B\,A$ and $\pi_2 : \mathcal{C}\,A \times B\,B$ such that for all objects C and morphisms $f : \mathcal{C}\,C\,A$ and $g : \mathcal{C}\,C\,B$, there exists a unique morphism $\langle f, g\rangle : \mathcal{C}\,C\,A \times B$ such that

$$\pi_1 \circ \langle f, g\rangle = f \text{ and } \pi_2 \circ \langle f, g\rangle = g$$

. Given a cartesian category, we define the following morphisms which will be used through the rest of the paper:

- $dup = \lambda x.(x, x)$ and $sdup = \lambda(x, y).((x, y), y)$
- $swap = \lambda(x, y).\,(y, x)$
- $assoc = \lambda((x, y), z).\,(x, (y, z))$ and $unassoc = \lambda(x, (y, z)).\,((x, y), z)$
- $perm = \lambda((x, y), z).\,((x, z), y)$
- $f \times g = \lambda(x, y).(f\,x, g\,y)$

3.2 Functors and Monads

A functor between a category $\mathcal{C}$ and a category $\mathcal{D}$ consists of a mapping of objects $F : \mathcal{C} \to \mathcal{D}$ and a mapping of morphisms $F : (A \to B) \to F\,A \to F\,B$ for each pair of objects A and B such that F preserves identities and compositions.

$$F\,id = id \tag{2a}$$
$$F\,(g \circ f) = F\,g \circ F\,f \tag{2b}$$

When $\mathcal{C} = \mathcal{D}$, the functor F is called an endofunctor. A monad in a category $\mathcal{C}$ consists of an endofunctor $M : \mathcal{C} \to \mathcal{C}$ and two indexed family of operators $return_A : A \to M\,A$ and $bind_{AB} : M\,A \to (A \to M\,B) \to M\,B$ for each pair of objects A and B. These morphisms must satisfy the following laws:

$$bind\,(return\,a)\,f = f\,a \tag{3a}$$
$$bind\,m\,return = m \tag{3b}$$
$$bind\,(bind\,f\,g)\,h = bind\,f\,(\lambda x.bind\,(g\,x)\,h) \tag{3c}$$

A monad in a category $\mathcal{C}$ gives rise to a Kleisli category $\mathcal{C}_M$ with has the same objects as $\mathcal{C}$, but with morphisms defined as $\mathcal{C}_M\,A\,B = \mathcal{C}\,A\,(M\,B)$ for each pair of objects A and B. Identities are given by $return$ and compositions are defined by $g \circ f = \lambda a.\ bind\,(f\,a)\,g$.

3.3 Arrows

An arrow in a cartesian category $\mathcal{C}$ consists of a mapping of objects $\text{Arr} : \mathcal{C} \to \mathcal{C} \to \mathcal{C}$ with three indexed families of operations:

$$\begin{array}{lcl} arr_{AB} & : & (A \to B) \to \text{Arr}\,A\,B \\ >\!\!>\!\!>_{ABC} & : & \text{Arr}\,A\,B \times \text{Arr}\,B\,C \to \text{Arr}\,A\,C \\ first_{ABC} & : & \text{Arr}\,A\,B \to \text{Arr}\,(A \times C)\,(B \times C) \end{array} \tag{4}$$

Arrows are elements of $\text{Arr}\,A\,B$ for each pair of objects A and B. The operations above must satisfy the laws enumerated in eq. (5), where we denote arrows by a, b, and c, and morphisms in the category $\mathcal{C}$ by f and g.

$$\text{arr}\,id >\!\!>\!\!> a = a \tag{5a}$$
$$a >\!\!>\!\!> \text{arr}\,id = a \tag{5b}$$
$$(a >\!\!>\!\!> b) >\!\!>\!\!> c = a >\!\!>\!\!> (b >\!\!>\!\!> c) \tag{5c}$$
$$\text{arr}\,(g \circ f) = \text{arr}\,f >\!\!>\!\!> \text{arr}\,g \tag{5d}$$
$$\text{first}\,a >\!\!>\!\!> \text{arr}\,\pi_1 = \text{arr}\,\pi_1 >\!\!>\!\!> a \tag{5e}$$
$$\text{first}\,a >\!\!>\!\!> \text{arr}\,(id \times f) = \text{arr}\,(id \times f) >\!\!>\!\!> \text{first}\,a \tag{5f}$$
$$\text{first}(\text{first}\,a) >\!\!>\!\!> \text{arr}\,assoc = \text{arr}\,assoc >\!\!>\!\!> \text{first}\,a \tag{5g}$$
$$\text{first}\,(\text{arr}\,f) = \text{arr}\,(f \times id) \tag{5h}$$
$$\text{first}(a >\!\!>\!\!> b) = \text{first}\,a >\!\!>\!\!> \text{first}\,b \tag{5i}$$

Each Arrow Arr on a category $\mathcal{C}$ gives rise to a category $\mathcal{C}_{\text{Arr}}$ with the same objects as $\mathcal{C}$ and morphisms defined by $\mathcal{C}_{\text{Arr}}\,A\,B = \text{Arr}\,A\,B$ for each pair of objects A and B. The identity of an object A is $\text{arr}\,id_A$ and composition is defined as $g \circ f = f >\!\!>\!\!> g$. This structure forms a category that follows eqs. (5a) to (5c). Additionally, eq. (5d) establishes that arr is an identity-on-objects functor from the initial category to the category of arrows. An arrow with loops refers to an arrow equipped with a feedback loop which satisfies additional laws. $loop_{ABC} : C \to \text{Arr}\,(A \times C)\,(B \times C) \to \text{Arr}\,A\,B$

$$\text{loop}\,c\,(\text{first}\,a >\!\!>\!\!>\ b) = a >\!\!>\!\!> \text{loop}\,c\,b \tag{6a}$$
$$\text{loop}\,c\,(a >\!\!>\!\!> \text{first}\,b) = \text{loop}\,c\,a >\!\!>\!\!> b \tag{6b}$$
$$\text{loop}\,c\,(\text{loop}\,d\,a) = \text{loop}\,(c,d)\,(\text{arr}\,unassoc >\!\!>\!\!> a >\!\!>\!\!> \text{arr}\,assoc) \tag{6c}$$

3.4 Bisimulations

Given an endofunctor F of a category C, a coalgebra for F (or a F-algebra) is a pair (X, f) where X and $f : X \to F\,X$ are an object and a morphism in the category $\mathcal{C}$, respectively. A bisimulation is a relation between elements of two coalgebras of a functor. Let $step_X : X \to F_{A,B}\,X$ and $step_Y : Y \to F_{A,B}\,Y$ be two coalgebras. A bisimulation between X and Y is a relation R such that for all $x \in X$ and $y \in Y$ with $R\,x\,y$, the following holds:

$$\text{whenever } step\ x\ a = (b, x'), \text{ there exists } y' \text{ such that } step\ y\ a = (b, y') \text{ and } R\ x'\ y'$$

Bisimilarity, denoted $x \sim y$, is the largest bisimulation between X and Y. Thus, to demonstrate that two elements are bisimilar, it suffices to exhibit a bisimulation that relates them. Bisimilarity between X and Y enjoys the following properties:

$$x \sim x \tag{7a}$$
$$x \sim y \Rightarrow y \sim x \tag{7b}$$
$$x \sim y \wedge y \sim z \Rightarrow x \sim z \tag{7c}$$

In particular, the relation $\sim$ is an equivalence relation when $X = Y$.

It is worth noting that this notion of bisimulation aligns more closely with simulation in the terminology of process algebras. Bisimulation requires that both elements simulates each other. However, when considering systems described by total functions, the two notions coincide. This holds true for the semantics of well-typed programs in the formalization of both YAMPA and MOLHOLES.

4 YAMPACORE

In this section, we present a formal semantics for the kernel language YAMPACORE, which captures the core concepts of YAMPA. This semantics is used in section 6, where we show how MOLHOLES programs, as defined in section 5, can be translated into YAMPACORE programs. Additionally, we demonstrate how bisimulation proofs can be used to establish that YAMPACORE effectively defines an arrow with loops. Furthermore, we show that any program can be transformed into an equivalent normal form. This well-known result follows directly from the equational theory of arrows with loops.

We define YAMPACORE as a two-level language: the first level consists of a host language whose semantics is provided by $\mathcal{C}_{\text{hst}}$, while the second level is a domain-specific language for stream processing. Programs represent total functions over streams of values, where streams are modeled as an indexed family of types $Stream\ A$, with A being a type in the host language. Notably, we do not assume that stream types are types of the host language, reflecting the fact that streams are not treated as first-class citizens in YAMPA. Stream types are defined by the following indexed family:

$$Stream\ A \quad = \quad \texttt{Cons} : A \times Stream\ A \to Stream\ A$$

The type $Stream\ A$ is isomorphic to the type of infinite lists of elements of type A. It serves as the carrier of a final coalgebra, characterized by $\lambda\,(\texttt{Cons}\ a\ l) = (a, l)$, for the functor $F_A\ X = A \times X$. A bisimulation over streams is defined as a relation R such that for all s and s' such that $R\ s\ s'$:

$$\text{if } s = \texttt{Cons}\ a\ s_1 \text{ then there exists } s_1' \text{ such that } s' = \texttt{Cons}\ a\ s_1' \text{ and } R\ s_1\ s_1'$$

Bisimilarity is defined, as usual, as the largest bisimulation.

4.1 Syntax and Semantics

We define the syntax of YAMPACORE using typed terms, which are elements of algebras $SF\ A\ B$, where A and B are types of the host language. The definition of SF aligns with the informal description of YAMPA provided in section 2.1.

$$\begin{aligned} SF \quad = \quad & \mid \texttt{Arr} : \forall A\ B.\ (A \to B) \to SF\ A\ B \\ & \mid \texttt{Comp} : \forall A\ B\ C.\ SF\ A\ B \times SF\ B\ C \to SF\ A\ C \\ & \mid \texttt{First} : \forall A\ B\ C.\ SF\ A\ B \to SF\ (A \times C)\ (B \times C) \\ & \mid \texttt{Loop} : \forall A\ B\ C.\ C \times SF\ (A \times C)\ (B \times C) \to SF\ A\ B \end{aligned}$$

For example, the term $\texttt{Loop}\ v\ (\texttt{Arr}\ swap)$ represents the signal function $delay\ v$ where $delay$ refers to the function defined in section 2.1. The semantic domain of terms is defined by the coinductive type:

$$sf\ A\ B = A \to (B \times sf\ A\ B)$$

which constitutes a terminal coalgebra $(sf\ A\ B, id)$ for the functor $F_{A,B}$, as introduced in section 3.

The stepwise semantics of terms is defined by the recursive function `step`. Its definition relies on the corecursive functions arr, $comp$, $first$ and $loop$, as detailed below.

$$
\begin{array}{lcl}
\texttt{step} & :: & SF\ A\ B \to sf\ A\ B \\
\texttt{step}\,(\texttt{Arr}\ f) & = & arr\ f \\
\texttt{step}\,(\texttt{Comp}\ sf_1\ sf_2) & = & comp\ (\texttt{step}\ sf_1)\ (\texttt{step}\ sf_2) \\
\texttt{step}\,(\texttt{First}\ sf) & = & first\ (\texttt{step}\ sf) \\
\texttt{step}\,(\texttt{Loop}\ sf\ v) & = & loop\ v\ (\texttt{step}\ sf) \\
 & & \\
arr\ f & = & \lambda x.(f\ x, arr\ f) \\
comp\ sf_1\ sf_2 & = & \lambda a.(c, comp\ sf_1'\ sf_2') \\
 & & \text{where } (b, sf_1') = sf_1\ a \text{ and } (c, sf_2') = sf_2\ b \\
first\ sf & = & \lambda(x,z).((y,z), first\ sf') \text{ where } y, sf' = sf\ x \\
loop\ v\ sf & = & \lambda x.(y, loop\ v'\ sf') \text{ where } (y, v'), sf' = sf\ (x, v)
\end{array}
$$

For example, the semantics of the term $\texttt{Loop}\,b\,(\texttt{Arr}\ swap)$ is given by

$$
\begin{array}{lcl}
\texttt{step}\,(\texttt{Loop}\,b\,(\texttt{Arr}\ swap)) & = & loop\,b\,(arr\ swap) \\
 & = & \lambda a.(b,\ loop\ a\ (arr\ swap)) \\
 & = & \lambda a.(b, \texttt{step}\,(\texttt{Loop}\,a\,(\texttt{Arr}\ swap)))
\end{array}
$$

It is worth noting that, in this example, the signal function produces a new signal function that is identical to the original, except for an updated internal state. In fact, this property holds for all YAMPACORE terms, where multiple updates may occur. This characteristic significantly simplifies bisimulation proofs. Finally, a term is lifted into a stream function by applying the corecursive function `run` to its stepwise semantics.

$$
\begin{array}{lcl}
\texttt{run} & : & sf\ A\ B \to Stream\ A \to Stream\ B \\
\texttt{run}\ sf\ (a : s) & = & b : s' \\
 & & \text{where } (b,\ sf') = sf\ a \wedge s' = \texttt{run}\ sf'\ s
\end{array}
$$

The functions arr, $comp$, $first$ and $loop$ define an arrow with loops on the category $\mathcal{C}_{\text{hst}}$. Furthermore, these functions are compatible with bisimilarity. To establish these results, we first refine the definition of bisimulation specifically for signal functions. A relation R is a bisimulation on $sf\ A\ B$ if for all sf_1 and sf_2 such that $R\ sf_1\ sf_2$:

$$\text{if } sf_1\ a = (b, sf_1') \text{ then there exists } sf_2' \text{ such that } sf_2\ a = (b, sf_2') \text{ and } R\ sf_1'\ sf_2'$$

As usual, we define bisimilarity, denoted $\cdot \sim \cdot$, as the largest bisimulation. This relation is an equivalence relation, as stated in section 3. We now proceed to prove the two results mentioned above.

Lemma 1. *Let sf, sf_1, sf_1', sf_2 and sf_2' be signal functions and let v be a value. The following properties hold:*

- *if $sf_1 \sim sf_1'$ and $sf_2 \sim sf_2'$ then $comp\ sf_1\ sf_2 \sim comp\ sf_1'\ sf_2'$*
- *if $sf \sim sf'$ then $first\ sf \sim first\ sf'$*
- *if $sf \sim sf'$ then $loop\ v\ sf \sim loop\ v\ sf'$*

Proof. The proof proceeds by bisimulation. For the first case, we define the relation R such as:

$$R = \{(comp\ sf_1\ sf_2,\ comp\ sg_1\ sg_2) \mid sf_1 \sim sg_1 \wedge sf_2 \sim sg_2\}$$

To prove the result it is sufficient to prove that R is a bisimulation. Suppose that $R\ sf\ sg$ and $sf\ a = (b, sf')$. We need to prove that:

$$\exists sg', sg\ a = (b, sg') \text{ and } R\ sf'\ sg'$$

By definition of R, there exists sf_1, sf_2, sg_1 and sg_2 such that $sf = comp\ sf_1\ sf_2$, $sg = comp\ sg_1\ sg_2$, $sf_1 \sim sg_1$ and $sf_2 \sim sg_2$. By definition of $comp$, there exists c, sf_1' and sf_2' such that $sf_1\ a = (c, sf_1')$, $sf_2\ c = (b, sf_2')$ and $sf' = comp\ sf_1'\ sf_2'$. By bisimilarity there exists sg_1' and sg_2' such that:

$$sg_1\ a = (c, sg_1') \text{ and } sf_1' \sim sg_1'$$
$$sg_2\ c = (b, sg_2') \text{ and } sf_2' \sim sg_2'$$

Thus taking $sg' = comp\ sg_1'\ sg_2'$, we have that $sg\ a = (b, sg')$ and $R\ sf'\ sg'$ holds. The other cases are similar, proving that both $\{first\ sf, first\ sg \mid sf \sim sg\}$ and $\{loop\ v\ sf, loop\ v\ sg \mid sf \sim sg\}$ are bisimulations. □

Theorem 1. *The functions arr, comp, first, and loop form an arrow with loops.*

Proof. For example, to prove that eq. (5g) holds, we have to prove that

$$comp\,(first\,(first\,sf))\,(arr\,assoc) \sim comp\,(arr\,assoc)\,(first\,sf)$$

First, define the relation

$$R_{ABC} = \{(comp\,(first\,(first\,sf))\,(arr\,assoc), comp\,(arr\,assoc)\,(first\,sf))\}_{sf:sf\,A\,B\,C}$$

Next, suppose that $(comp\,(first\,(first\,sf))\,(arr\,assoc))\,((a_1, a_2), a_3) = ((b, (a_2', a_3')), sg)$ for some b, a_2', a_3' and sg. By definition of the functions *comp*, *first* and *arr* we have $a_2' = a_2$, $a_3' = a_3$ and $sf\,a = (b, sf')$ and $sg = comp\,(first\,(first\,sf'))\,(arr\,assoc)$ for some sf'. We also have

$$(comp\,(arr\,assoc)\,(first\,sf))\,((a_1, a_2), a_3) = ((b, (a_2, a_3)), sg')$$

where $sg' = comp\,(arr\,assoc)\,(first\,sf)$. By definition of R_{ABC}, we have $R_{ABC}\,sg\,sg'$. Other cases are similar. Take the relation that connects the left and right sides of the equation, generalized over all variables. □

As stated in section 3, this arrow gives rise to a category $\mathcal{C}_{sf}$, where arr is an identity-on-objects functor between $\mathcal{C}_{\text{hst}}$ and $\mathcal{C}_{sf}$. Together, these results enable us to establish the correctness of multiple transformations in the following section.

4.2 Normal form

We demonstrate that any YAMPACORE program can be transformed into a normal form $\texttt{Loop}\,(v, \texttt{Arr}\,f)$ for some value v and function f of the host language. Building on bisimulation, we define the relation $\equiv$ over YAMPACORE terms as $t_1 \equiv t_2$ iff $\texttt{step}\,t_1 \sim \texttt{step}\,t_2$. By lemma 1, we can derive that $\equiv$ is a congruence on the term algebras $SF\,A\,B$.

Lemma 2. *Let t, t_1, t_1', t_2 and t_2' be terms. The following properties hold:*

- *if $t_1 \equiv t_1'$ and $t_2 \equiv t_2'$ then* $\texttt{Comp}\,(t_1, t_2) \equiv \texttt{Comp}\,(t_1', t_2')$
- *if $t \equiv t'$ then* $\texttt{First}\,t \equiv \texttt{First}\,t'$
- *if $t \equiv t'$ then* $\texttt{Loop}\,(v, t) \equiv \texttt{Loop}\,(v, t')$

Proof. Immediate by definition of $\equiv$ and lemma 1. □

We now have all the necessary elements to prove that any YAMPACORE term can be transformed into a normal form.

Theorem 2. *For every term t, there exists a value v and a function f in the host language such that* $t \equiv \texttt{Loop}\,(v, \texttt{Arr}\,f)$

Proof. The proof is by structural induction on the term t and by definition of $\equiv$.

- $\texttt{Arr}\,f \equiv \texttt{Loop}\,(\texttt{tt}, \texttt{Arr}\,(\lambda(x, y).(f\,x, \texttt{tt})))$
- if $t \equiv \texttt{Loop}\,(v, \texttt{Arr}\,f)$ then $\texttt{First}\,t \equiv \texttt{Loop}\,(v, \texttt{Arr}\,g)$ where
$$g = \lambda((a, d), c).\texttt{let}\,(b, c') = f\,(a, c)\,\texttt{in}\,((b, d), c')$$
- if $t_1 \equiv \texttt{Loop}\,(d, \texttt{Arr}\,f_1)$ and $t_2 \equiv \texttt{Loop}\,(e, \texttt{Arr}\,f_2)$ then $\texttt{Comp}\,(t_1, t_2) \equiv \texttt{Loop}\,((d, e), \texttt{Arr}\,g)$ where
$$g = \lambda(a, (c_1, c_2)).\texttt{let}\,(b, c_1') = f_1\,(a, c_1)\,\texttt{in}\,\texttt{let}\,(c, c_2') = (b, c_2)\,\texttt{in}\,(c, (c_1', c_2'))$$
- if $t \equiv \texttt{Loop}\,(d, \texttt{Arr}\,f)$ then $\texttt{Loop}\,(c, t) \equiv \texttt{Loop}\,((c, d), \texttt{Arr}\,g)$ where
$$g = \lambda(a, (c, d)).\texttt{let}\,((b, c'), d') = f\,((a, c), d)\,\texttt{in}\,(b, (c', d'))$$

In each case, bisimilarity is established at the semantics level by considering the relation that connects the left and right sides of the equation, generalized over all variables, and ensuring that the sub-expressions are bisimilar. For example, in the case of $first$, take the relation $R = \{(\mathit{first}\ sf, \mathit{loop}\ d\ (\mathit{arr}\ g)) \mid sf \sim \mathit{loop}\ d\ (\mathit{arr}\ f)\}$, where g is defined as above, and prove that it is a bisimulation. We pose x, z two values from the host language such as :

$$(\mathit{first}\ sf)\ (x, z) = ((y, z), \mathit{first}\ sf') \text{ where } (y, sf') = sf\ x$$

We know that $sf \sim \mathit{loop}\ v\ (\mathit{arr}\ f)$, then $(\mathit{loop}\ v\ (\mathit{arr}\ f))\ x = (y, sf'')$ and $R\ sf'\ sf''$. Knowing that $\mathit{arr}\ f$ is an invariant during evaluation, we can deduce that there exists v' such as :

$$(\mathit{loop}\ v\ (\mathit{arr}\ f))\ x = (y, \mathit{loop}\ v'\ (\mathit{arr}\ f)) \text{ where } (y, v') = f\ (x, v)$$

Then, there exists $\mathit{loop}\ v'\ (\mathit{arr}\ g)$ such that $\mathit{loop}\ v\ (\mathit{arr}\ g)\ (x, z) = ((y, z), \mathit{loop}\ v'\ (\mathit{arr}\ g))$ which conclude this case. □

5 The MOLHOLES language

This section introduces MOLHOLES, a simple reactive functional language that extends WORMHOLES. Like WORMHOLES, this language replaces the loop combinators with signal functions to read and write resources. Unlike WORMHOLES we do not introduce a binder for local resources, instead we rely on the status of resources which are classified as inputs, outputs or internal resources. Input and output resources are single-use, meaning they must be accessed exactly once. Internal resources, on the other hand, can be read from and written to at any time. The language is equipped with a type system, presented in section 5.4, that ensures that input and output resources are used correctly.

5.1 Syntax

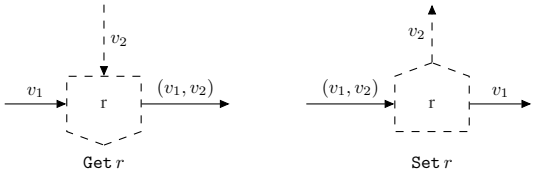


Figure 4: Graphical representation of resource accesses

The syntax of MOLHOLES is presented in fig. 5. The type $\mathit{Ref}\ A$ represents resources that carry a value of type A and are uniquely identified by a natural number. A value of type $\mathit{Ref}\ A$ is expressed as `Ref` $A\ n$, where n is the resource identifier. Terms constructed with `Arr`, `First` and `Comp` are similar to their counterparts in YAMPA. The `Get` and `Set` constructs are used to read from and write to resources, respectively. A Graphical representation is provided in fig. 4.

- `Get` r represents a signal function that takes an input x and returns a pair (x, y), where y is the value of the resource r.
- `Set` r represents a signal function that takes an input (x, y) and returns x, after updating resource r with the value y

$$\begin{aligned}
\mathit{Ref} \quad &= \quad \texttt{Ref} : \forall A.\ A \to \mathit{Nat} \to \mathit{Ref}\ A \\
\mathit{RSF} \quad &= \quad \mid \texttt{Arr} : \forall A\ B.\ (A \to B) \to \mathit{RSF}\ A\ B \\
&\quad\ \mid \texttt{First} : \forall A\ B\ C.\ \mathit{RSF}\ A\ B \to \mathit{RSF}\ (A \times C)\ (B \times C) \\
&\quad\ \mid \texttt{Comp} : \forall A\ B\ C.\ \mathit{RSF}\ A\ B \times \mathit{RSF}\ B\ C \to \mathit{RSF}\ A\ C \\
&\quad\ \mid \texttt{Get} : \forall A\ B.\ \mathit{Ref}\ B \to \mathit{RSF}\ A\ (A \times B) \\
&\quad\ \mid \texttt{Set} : \forall A\ B.\ \mathit{Ref}\ B \to \mathit{RSF}\ (A \times B)\ A
\end{aligned}$$

Figure 5: Syntax of MOLHOLES

We adopt a programming model in which all interactions with the environment are mediated through resources. Consequently, both input and output types of entire programs are restricted to the singleton type Unit, whose only value

is denoted as `tt`. A program is defined by a term of type $RSF\ Unit\ Unit$ along with mappings that associate its input and output resources with types and its internal resources with initial typed values.

$$\begin{array}{rcll} Prog & = & \{ & \texttt{inputs} : List\ Type; \\ & & & \texttt{outputs} : List\ Type; \\ & & & \texttt{internals} : List\, Val_{Type}; \\ & & & \texttt{program} : RSF\ Unit\ Unit\ \} \end{array}$$

where $Type$ represents the types of the host language and Val_{Type} represents typed values in the host language.

Example 1. *The delay function presented in section 3 can be expressed in* MOLHOLES *as follows:*

$$\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\,r, \texttt{Arr}\,(\lambda(x,y).(y,x))), \texttt{Set}\,r) : RSF\ A\ A$$

where for r : $Ref\ A$, the first value of the output stream is the initial value of r. A complete example is the program below, which generates the sequence of natural numbers:

$$\begin{array}{ll} \{ & \texttt{inputs} = []; \\ & \texttt{outputs} = [Nat]; \\ & \texttt{internals} = [0 : Nat]; \\ & \texttt{program} = \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\,r_0, \texttt{Arr}\,(\lambda((),n).(((),n),n+1))), \texttt{Set}\,r_0), \texttt{Set}\,r_1)\} \end{array}$$

where r is the unique input resource and r_0 is the unique internal resource.

5.2 Dynamic semantics

The semantic domain of stepwise computations in MOLHOLES is the Kleisli category associated with a state monad that manages memory. This memory accounts for different types of resources and their access rights, making the semantics a partial function. However, the type system of section 5.4 ensures that this function is total for well-typed programs. The memory domain and the monad built on top of it are introduced in section 5.2.1 and section 5.2.2, respectively. The stepwise semantics domain is formalized in section 5.3 as the Kleisli category associated with the state monad. Similarly to what we have done for YAMPACORE, we demonstrate that this semantic domain forms an arrow. We also introduce a set of equations satisfied by the new constructors, `Get` and `Set`. These equations serve as the counterpart to the laws governing loops in YAMPACORE. Finally, the stream-based semantics of programs is formalized in section 5.3.2.

5.2.1 Memory

Memory is represented as elements of type $Memory$, defined as partial mappings (denoted by $\rightharpoonup$) from resource identifiers to cells (see fig. 6). A cell, of type $Cell$, is a pair consisting of a tag of type Tag and an optional value of the type specified by the tag. Given a type A we denote by $A^?$ the type A extended with an additional element `undef`. The tag itself is a pair containing a status, which indicates the accessibility of the resource, and a type, which specifies the type of values carried by the resource. The status of a resource can take one of the following forms :

- `Internal`: Represents an internal resource, which can be read from and written to at any time.
- `Input` b: Represents an input resource, which can only be read from if the boolean b is true.
- `Output` b: Represents an output resource, which can only be written if the boolean b is true.

The predicates `readable` and `writable` determine whether a resource is readable or writable within a given memory. The functions `read` r and `write` $r\ v$ are total over memories satisfying `readable` r and `writable` r, respectively.

$$\begin{array}{rcl} \texttt{readable}\,(\texttt{Ref}\,A\,n)\,\sigma & = & (\exists v : A.\,\sigma\,n = \texttt{Cell}\,(\texttt{Internal}, A)\,v) \vee \\ & & \quad (\exists v : A.\,\sigma\,n = \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, A)\,v) \\ \texttt{writable}\,(\texttt{Ref}\,A\,n)\,\sigma & = & (\exists v : A.\,\sigma\,n = \texttt{Cell}\,(\texttt{Internal}, A)\,v) \vee \\ & & \quad (\sigma\,n = \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, A)\,\texttt{undef}) \end{array}$$

Note that in the definitions of `read` and `write` we require that the type of the value stored in the cell matches the type of the resource. This may not always be the case, as the memory is only aware of the identifier of the resource and not its type. For well-typed programs, the two types are always identical.

$$
\begin{array}{lcl}
\mathit{Status} & = & \mid \texttt{Internal} : \mathit{Status} \\
 & & \mid \texttt{Input} : \mathit{Bool} \to \mathit{Status} \\
 & & \mid \texttt{Output} : \mathit{Bool} \to \mathit{Status} \\
\mathit{Tag} & = & \mathit{Status} \times \mathit{Type} \\
\mathit{Cell} & = & \texttt{Cell} : \forall t : \mathit{Tag}.(\mathit{snd}\, t)^? \to \mathit{Cell} \\
\mathit{Memory} & = & \mathit{Nat} \rightharpoonup \mathit{Cell}
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{read} & : & \mathit{Ref}\ A \to \mathit{Memory} \rightharpoonup A \times \mathit{Memory} & \\
\texttt{read}\, r\, \sigma & = & \texttt{case}\, \sigma\, (\texttt{id}\, r)\, \texttt{of} & \\
 & & \mid \texttt{Cell}\, (\texttt{Internal}, B)\, x \Rightarrow (x, \sigma) & \text{if } A = B \\
 & & \mid \texttt{Cell}\, (\texttt{Input}\, \texttt{true}, B)\, x & \\
 & & \quad \Rightarrow (x, \sigma[(\texttt{id}\, r) \mapsto \texttt{Cell}\, (\texttt{Input}\, \texttt{false}, B)\, \texttt{undef}]) & \text{if } A = B
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{write} & : & \mathit{Ref}\ A \to \mathit{Memory} \to A \rightharpoonup \mathit{Memory} & \\
\texttt{write}\, r\, \sigma\, v & = & \texttt{case}\, \sigma\, (\texttt{id}\, r)\, \texttt{of} & \\
 & & \mid \texttt{Cell}\, (\texttt{Internal}, B)\, x & \\
 & & \quad \Rightarrow \sigma[(\texttt{id}\, r) \mapsto \texttt{Cell}\, (\texttt{Internal}, B)\, v] & \text{if } A = B \\
 & & \mid \texttt{Cell}\, (\texttt{Output}\, \texttt{true}, B)\, \texttt{undef} & \\
 & & \quad \Rightarrow \sigma[(\texttt{id}\, r) \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{false}, B)\, v] & \text{if } A = B \\
\multicolumn{4}{l}{\text{where } \sigma[n \mapsto c] = \lambda n'.\, \texttt{if}\ n = n'\ \texttt{then}\ c\ \texttt{else}\ \sigma\, n'}
\end{array}
$$

Figure 6: Memory domain

5.2.2 State Monad

Building on the type Memory, we define a state monad based on the functor

$$
\begin{array}{lcl}
\mathit{St}\, A & = & \mathit{Memory} \rightharpoonup A \times \mathit{Memory} \\
\mathit{St}\, f & = & \lambda h\, \sigma.(f\, a, \sigma') \text{ where } (a, \sigma') = h\, \sigma
\end{array}
$$

This monad is equipped with the following operations:

$$
\begin{array}{lcllcl}
\texttt{return}_A & : & A \to \mathit{St}\, A & \texttt{bind}_{AB} & : & \mathit{St}\, A \to (A \to \mathit{St}\, B) \to \mathit{St}\, B \\
\texttt{return}\, a & = & \lambda\sigma.(a, \sigma) & \texttt{bind}\, m\, f & = & \lambda\sigma.f\, x\, \sigma' \text{ where } (x, \sigma') = m\, \sigma \\
 & & & & & \\
\texttt{get}_A\, r & : & \mathit{Ref}\ A \to \mathit{St}\, A & \texttt{set}_A\, r\, v & : & \mathit{Ref}\ A \to A \to \mathit{St}\ \mathit{Unit} \\
\texttt{get}\, r & = & \texttt{read}\, r & \texttt{set}\, r\, v & = & \lambda\sigma.(\texttt{tt}, \texttt{write}\, r\, \sigma\, v)
\end{array}
$$

The functions $\texttt{get}\, r$ and $\texttt{set}\, r\, v$ are total over memories satisfying the predicates $\texttt{readable}\, r$ and $\texttt{writable}\, r$, respectively. These operations satisfy the monad laws presented in eqs. (3a) to (3c). We extend this equational theory with laws that establish relationships between the `get` and `set` operations. The validity of these laws for the state monad follows directly from the definition of the memory domain. Only internal resources are concerned by eqs. (8a) to (8e). These rules allow the removal of certain resources accesses, which is not permitted for input and output resources, as their accesses update their status (readable or writable). Consequently, equations eqs. (8a) to (8e) should be interpreted as follows: $f = g$ means $f\, \sigma = g\, \sigma$ for all σ where the resource r is an internal resource. Similarly, other rules apply to all types of resources, provided that for read (resp. write) access, the resource is readable (resp. writable).

$$\texttt{bind}\, (\texttt{get}\, r)\, (\lambda x.\texttt{get}\, r) = \texttt{get}\, r \tag{8a}$$

$$\texttt{bind}\, (\texttt{get}\, r)\, (\texttt{set}\, r) = \texttt{return}\, \texttt{tt} \tag{8b}$$

$$\texttt{bind}\, (\texttt{set}\, r\, x)\, (\lambda\texttt{tt}.\, \texttt{get}\, r) = \texttt{bind}\, (\texttt{set}\, r\, x)\, (\lambda\texttt{tt}.\, \texttt{return}\, x) \tag{8c}$$

$$\texttt{bind}\, (\texttt{set}\, r\, x)\, (\lambda\texttt{tt}.\texttt{set}\, r\, y) = \texttt{set}\, r\, y \tag{8d}$$

$$\texttt{bind}\, (\texttt{get}\, r)\, (\lambda y.\texttt{return}\, x) = \texttt{return}\, x \tag{8e}$$

$$\texttt{bind}\, (\texttt{get}\, r)\, (\lambda x.\texttt{get}\, r') = \texttt{bind}\, (\texttt{get}\, r')\, (\lambda x.\texttt{bind}\, (\texttt{get}\, r)\, (\lambda y.\texttt{return}\, x)) \tag{8f}$$

$$\texttt{bind}\, (\texttt{get}\, r)\, (\lambda x.\texttt{set}\, r'\, y) = \texttt{bind}\, (\texttt{set}\, r'\, y)\, (\lambda\texttt{tt}.\texttt{bind}\, (\texttt{get}\, r)\, (\lambda x.\texttt{return}\, \texttt{tt})) \tag{8g}$$

$$\texttt{bind}\, (\texttt{set}\, r\, x)\, (\lambda\texttt{tt}.\texttt{get}\, r') = \texttt{bind}\, (\texttt{get}\, r')\, (\lambda y.\texttt{bind}\, (\texttt{set}\, r\, x)\, (\lambda\texttt{tt}.\texttt{return}\, y)) \tag{8h}$$

$$\texttt{bind}\, (\texttt{set}\, r\, x)\, (\lambda\texttt{tt}.\texttt{set}\, r'\, y) = \texttt{bind}\, (\texttt{set}\, r'\, y)\, (\lambda\texttt{tt}.\texttt{set}\, r\, x) \tag{8i}$$

Lemma 3. *The functor St, along with the operations defined above, satisfies the monad laws presented in* eqs. (3a) to (3c) *and the additional laws given in* eqs. (8a) to (8d) and (8f) to (8i)*.*

Proof. The proof follows directly from the definitions of the operators and the memory domain. In each case, expanding the definition reduces the proof to a property of the memory domain. For instance, for eq. (8b), assuming the resource is internal we must show that if $\texttt{read}\, r\, \sigma = (a, \sigma')$ then $\sigma' = \sigma$ and $\texttt{write}\, r\, \sigma\, a = \sigma$. This holds because accesses to an internal resource do not affect access rights. Other cases follow similarly. □

Example 2. *By combining these equations, several equivalences can be derived. For example, applying eq.* (3b)*, eq.* (8e) *and eq.* (8f) *to an internal resource r allows the removal of an unused read access.*

$$\begin{aligned} \texttt{bind}\,(\texttt{get}\, r)\,(\lambda x.\texttt{get}\, r') &= \texttt{bind}\,(\texttt{get}\, r')\,(\lambda x.\texttt{bind}\,(\texttt{get}\, r)\,(\lambda y.\texttt{return}\, x)) \\ &= \texttt{bind}\,(\texttt{get}\, r')\,\texttt{return} \\ &= \texttt{get}\, r' \end{aligned}$$

5.3 Semantic domain

Now that we have all the necessary elements, we can define the semantic domain of MOLHOLES and prove that it satisfies the arrow laws, as well as additional laws similar to those governing loops in YAMPACORE. Together, these laws enable us to prove the correctness of the transformations presented in the next section. The semantic domain of combinators is defined by the type

$$rsf\ A\, B = A \to St\, B$$

Notably, unlike the definition of the semantic domains $sf\ A\, B$, the definition of $rsf\ A\, B$ is not corecursive. As a result, program transformations within MOLHOLES are greatly simplified. Only the final transformation to YAMPA will require the use of bisimulation. To achieve this we lift this semantics domain to a coalgebra of the functors $F_{A,B}$.

Building on this definition, we define the category $\mathcal{C}_{rsf}$ as the Kleisli category associated with the monad St. This category has the types of the host language as objects and morphisms of type $rsf\ A\, B$. As stated in section 3, identity morphisms and compositions are given by `return` and `bind`, respectively. Their definitions are restated below.

$$\begin{aligned} id &= \lambda x.\, \texttt{return}\, x \\ (g \circ f) &= \lambda x.\, \texttt{bind}\,(f\, x)\, g \end{aligned}$$

Next, we define the following functions, where the arr function defines identity-on-objects functors between $\mathcal{C}_{\text{hst}}$ and $\mathcal{C}_{rsf}$.

$$\begin{aligned} arr\ f &= \lambda x.\, \texttt{return}\,(f\, x) \\ comp\ rsf_1\ rsf_2 &= rsf_2 \circ rsf_1 \\ first\ rsf &= \lambda(x, c).\, \texttt{bind}\,(rsf\, x)\,(\lambda y.\, \texttt{return}\,(y, c)) \\ get\ r &= \lambda x.\, \texttt{bind}\,(\texttt{get}\, r)\,(\lambda y.\, \texttt{return}\,(x, y)) \\ set\ r &= \lambda(x, y).\, \texttt{bind}\,(\texttt{set}\, r\, y)\,(\lambda \texttt{tt}.\, \texttt{return}\, x) \end{aligned}$$

Theorem 3. *The functions arr, $first$, and $comp$ form an arrow.*

Proof. The proof follows by simple equational reasoning. We prove that eq. (5d) holds. By definition of arr and $comp$, we have $arr\,(g \circ f) = \lambda x.\texttt{return}\,(g\,(f\, x))$ and $arr\ f \circ arr\ g = \lambda x.\texttt{bind}\,(\texttt{return}\,(f\, x))\,(arr\, g)$. Applying eq. (3a) in the right-hand side, we obtain the left-hand side. Other cases are similar, using the monad laws for St. □

In addition to arrow laws, the get and set function bring new equations defined in eqs. (9) to (11). Those equations play a role similar to the laws governing loops in YAMPACORE but are separated into three groups. The first group governs the reordering of get operations, allowing them to be moved to the left or eliminated.

$$arr\ f >\!\!>\!\!> get\ r = get\ r >\!\!>\!\!> first\ arr\ f \tag{9a}$$

$$first\,(get\ r >\!\!>\!\!> rsf) = get\ r >\!\!>\!\!> arr\ perm >\!\!>\!\!> first\ rsf \tag{9b}$$

$$first\,(rsf) >\!\!>\!\!> get\ r = first\,(rsf >\!\!>\!\!> get\ r) >\!\!>\!\!> arr\ perm \tag{9c}$$

$$get\ r >\!\!>\!\!> get\ r' = get\ r' >\!\!>\!\!> get\ r >\!\!>\!\!> arr\ perm \tag{9d}$$

$$get\ r >\!\!>\!\!> get\ r = get\ r >\!\!>\!\!> arr\ sdup \tag{9e}$$

Similarly, the second groups governs the reordering of set operations, allowing them to be moved to the right or eliminated.

$$set\, r >\!\!>\!\!> arr\, f = first\, arr\, f >\!\!>\!\!> set\, r \tag{10a}$$
$$first\, (rsf >\!\!>\!\!> set\, r) = first\, rsf >\!\!>\!\!> arr\, perm >\!\!>\!\!> set\, r \tag{10b}$$
$$set\, r >\!\!>\!\!> first\, (rsf) = arr\, perm >\!\!>\!\!> first\, (set\, r >\!\!>\!\!> rsf) \tag{10c}$$
$$set\, r >\!\!>\!\!> set\, r' = arr\, perm >\!\!>\!\!> set\, r' >\!\!>\!\!> set\, r \tag{10d}$$
$$set\, r >\!\!>\!\!> set\, r = arr\, fst >\!\!>\!\!> set\, r \tag{10e}$$

Finally, the third group governs the reordering of consecutive set and get operations. eqs. (11b) and (11c) apply only when the resource is internal, while eq. (11a) holds for any resource.

$$set\, r >\!\!>\!\!> get\, r' = get\, r' >\!\!>\!\!> arr\, perm >\!\!>\!\!> set\, r \tag{11a}$$
$$set\, r >\!\!>\!\!> get\, r = arr\, sdup >\!\!>\!\!> set\, r \tag{11b}$$
$$get\, r >\!\!>\!\!> set\, r = arr\, id \tag{11c}$$

Those laws are satisfied, thus forming an arrow with resources that mimic arrows with loops.

Lemma 4. eqs. (9a) to (9d), eqs. (10a) to (10d) *and* eq. (11a) *hold for any resources.* eqs. (9e), (10e), (11b) and (11c) *hold for internal resources.*

Proof. The proof follows by simple equational reasoning, using the definitions, the monad laws eqs. (3a) to (3c), and the state monad laws eqs. (8a) to (8i). We prove that eq. (9a) holds, other cases follows similarly.

$$\begin{array}{lll}
 & get\, r >\!\!>\!\!> first\, arr\, f & \\
= & \lambda x.\mathtt{bind}\,(\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.\mathtt{return}\,(x,y)))\,(first\, arr\, f) & (def.) \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.\mathtt{bind}\,(\mathtt{return}\,(x,y))\,(first\, arr\, f)) & (eq.\ (3c)) \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.(first\, arr\, f)\,(x,y)) & (eq.\ (3a)) \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.(\mathtt{bind}\,(\mathtt{return}\,(f\, x))\,(\lambda z.\mathtt{return}(z,y)))) & (def.) \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.(\mathtt{return}\,(f\, x,y))) & (eq.\ (3a)) \\
= & \lambda x.get\, r\,(f\, x) & (def.) \\
= & \lambda x.\mathtt{bind}\,(\mathtt{return}(f\, x))\, get\, r & (eq.\ (3a)) \\
= & arr\, f >\!\!>\!\!> get\, r & (def.)
\end{array}$$

□

5.3.1 Stepwise evaluation

The functions defined previously enable a simple recursive definition of the stepwise semantics of MOLHOLES terms. This semantics maps a term of type $RSF\, A\, B$ to a morphism in $\mathcal{C}_{rsf}$.

$$\begin{array}{lcl}
\mathtt{step} & : & RSF\, A\, B \to rsf\, A\, B \\
\mathtt{step}\,(\mathtt{Arr}\, f) & = & arr\, f \\
\mathtt{step}\,(\mathtt{Comp}\,(rsf_1, rsf_2)) & = & comp\,(\mathtt{step}\, rsf_1)\,(\mathtt{step}\, rsf_2) \\
\mathtt{step}\,(\mathtt{First}\, rsf) & = & first\,(\mathtt{step}\, rsf) \\
\mathtt{step}\,(\mathtt{Get}\, r) & = & get\, r \\
\mathtt{step}\,(\mathtt{Set}\, r) & = & set\, r
\end{array}$$

The relation $\equiv$ over MOLHOLES terms is defined as $t \equiv t'$ iff $\mathtt{step}\, t = \mathtt{step}\, t'$. This relation is a congruence, as stated by the following straightforward lemma.

Lemma 5. *Let t, t_1, t_1', t_2, t_2' be terms of* MOLHOLES. *The following properties hold:*

- *if $t_1 \equiv t_1'$ and $t_2 \equiv t_2'$ then* $\mathtt{Comp}\,(t_1, t_2) \equiv \mathtt{Comp}\,(t_1', t_2')$
- *if $t \equiv t'$ then* $\mathtt{First}\ t \equiv \mathtt{First}\ t'$

Proof. This follows immediately from the definition of $\equiv$. □

5.3.2 Stream functions

The semantics of a program p has type $\mathit{Stream}\,(\alpha\,(\texttt{inputs}\,p)) \to \mathit{Stream}\,(\alpha\,(\texttt{outputs}\,p))$ where α is the function that converts a list into a tuple. It is given by $\texttt{run}\,p\,(\texttt{init}\,p)$ where run is defined as follows:

$$\begin{array}{lcl} \texttt{run}\,(p : \mathit{Prog}) & : & \mathit{Memory} \to \mathit{Stream}\,(\alpha\,(\texttt{inputs}\,p)) \rightharpoonup \mathit{Stream}\,(\alpha\,(\texttt{outputs}\,p)) \\ \texttt{run}\,p\,\sigma\,(\texttt{Cons}\,i\,s) & = & \texttt{Cons}\,(\texttt{push}\,p\,\sigma')\,(\texttt{run}\,p\,\sigma'\,s) \\ & & \text{where } (\texttt{tt}, \sigma') = \mathit{step}\,(\texttt{program}\,p)\,\texttt{tt}\,(\texttt{pull}\,p\,\sigma\,i) \end{array}$$

where, noting $k\,p = |\texttt{internals}\,p|$, $k_{in}\,p = |\texttt{inputs}\,p|$ and $k_{out}\,p = |\texttt{outputs}\,p|$, the functions init, pull and push are defined as follows:

$$\begin{array}{lcl} \texttt{init} & : & \mathit{Prog} \to \mathit{Memory} \\ \texttt{init}\,p\,n & = & \begin{cases} \texttt{Cell}\,(\texttt{Internal}, \tau)\,v & \text{if } k_{in}\,p + k_{out}\,p \le n < k_{in}\,p + k_{out}\,p + k\,p \\ & \text{and } v : \tau = (\texttt{internal}\,p)_{n-(k_{in}\,p + k_{out}\,p)} \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

$$\begin{array}{lcl} \texttt{pull} & : & \mathit{Prog} \to \mathit{Memory} \to (\mathit{Val}_{\mathit{Type}} \times \ldots) \times \mathit{Val}_{\mathit{Type}} \to \mathit{Memory} \\ \texttt{pull}\,p\,\sigma\,i\,n & = & \begin{cases} \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, \tau)\,v & \text{if } n < k_{in}\,p, \\ & \text{and } \tau = (\texttt{inputs}\,p)_n \\ & \text{and } v : \tau = (\texttt{split}\,i\,(k_{in}\,p))_n \\ \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, \tau)\,\texttt{undefined} & \text{if } k_{in}\,p \le n < k_{in}\,p + k_{out}\,p \\ & \text{and } \tau = (\texttt{outputs}\,p)_{n-(k_{in}\,p)} \\ \sigma\,n & \text{otherwise} \end{cases} \end{array}$$

$$\begin{array}{lcl} \texttt{split}\,p\,n & : & (\mathit{Val}_{\mathit{Type}} \times \ldots) \to \mathit{Nat} \to \mathit{List}\ \mathit{Val}_{\mathit{Type}} \\ \texttt{split}\,p\,n & & \begin{cases} [p] & \text{if } n = 0 \\ [] & \text{if } p = \texttt{tt} \\ (\texttt{split}\,p'\,m) +\!\!+ [v] & \text{if } n = m + 1 \text{ and } p = (p', v) \end{cases} \end{array}$$

$$\begin{array}{lcl} \texttt{push} & : & \mathit{Prog} \to \mathit{Memory} \to (\mathit{Val}_{\mathit{Type}} \times \ldots) \times \mathit{Val}_{\mathit{Type}} \\ \texttt{push}\,p\,\sigma & = & \alpha\ [v \mid\ \forall n, k_{in}\,p \le n < k_{in}\,p + k_{out}\,p \text{ and } \sigma\,n = \texttt{Cell}\,(\texttt{Output}\,\texttt{false}, \tau)\,v\ ] \end{array}$$

The function run is partial only because the function step is partial. Therefore, it is total for well-typed programs, as defined in the next section.

5.4 Static semantics

We define a simple static semantics for MOLHOLES, ensuring the correct use of memory. Specifically, it guarantees that in well-typed programs, each input resource is read exactly once and each output resource is written exactly once.

5.4.1 Abstract domain

The abstract domain consists of an abstract memory, which maps resource identifiers to abstract memory cells. These memory cells are simply tags, as defined in the previous section. Models of an abstract memory Σ correspond to the set $\gamma\,\Sigma$ of concrete memories that share the same domain as Σ and where the corresponding concrete and abstract cells are also related. The abstraction relation over memory cells imposes that the tags is the same and that the value held by the concrete memory has the correct type when the tag indicates that it must be defined.

$$\mathit{Memory}^\dagger = \mathit{Nat} \rightharpoonup \mathit{Tag}$$

$$\begin{array}{l} \gamma\,(st, \tau) = \{\texttt{Cell}\,(st, \tau)\,v \mid \text{if } st \in \texttt{defined} \text{ then } v \text{ is a value of type } \tau \text{ else undefined}\} \\ \text{where } \texttt{defined} = \{\texttt{Internal}, \texttt{Input}\,\texttt{true}, \texttt{Output}\,\texttt{false}\} \end{array}$$

$$\gamma\,\Sigma = \{\sigma \mid \texttt{dom}(\sigma) = \texttt{dom}(\Sigma) \wedge \forall n. n \in \texttt{dom}(\sigma) \to \sigma\,n \in \gamma\,(\Sigma\,n)\}$$

We define abstract $\texttt{read}^\dagger$ and $\texttt{write}^\dagger$ operations, along with the predicates $\texttt{readable}^\dagger$ and $\texttt{writable}^\dagger$, which ensure that these operations are well-defined. These operations update the status of memory cells, reflecting the fact that

access has occurred.

$$
\begin{array}{lcl}
\texttt{read}^\dagger & : & \mathit{Ref}\ A \to \mathit{Memory}^\dagger \rightharpoonup \mathit{Memory}^\dagger \\
\texttt{read}^\dagger\, r\, \Sigma & = & \texttt{case}\, \Sigma\, (\texttt{id}\, r)\, \texttt{of} \\
 & & \quad \mid (\texttt{Internal}, B) \Rightarrow \Sigma \\
 & & \quad \mid (\texttt{Input}\, \texttt{true}, B) \Rightarrow \Sigma\, [\texttt{id}\, r \mapsto (\texttt{Input}\, \texttt{false}, B)]
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{write}^\dagger & : & \mathit{Ref}\ A \to \mathit{Memory}^\dagger \rightharpoonup \mathit{Memory}^\dagger \\
\texttt{write}^\dagger\, r\, \Sigma & = & \texttt{case}\, \Sigma\, (\texttt{id}\, r)\, \texttt{of} \\
 & & \quad \mid (\texttt{Internal}, B) \Rightarrow \Sigma \\
 & & \quad \mid (\texttt{Output}\, \texttt{true}, B) \Rightarrow \Sigma[\texttt{id}\, r \mapsto (\texttt{Output}\, \texttt{false}, B)]
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{readable}^\dagger\, (\texttt{Ref}\, A\, n)\, \Sigma & = & \Sigma\, n = (\texttt{Internal}, A) \vee \Sigma\, n = (\texttt{Input}\, \texttt{true}, A) \\
\texttt{writable}^\dagger\, (\texttt{Ref}\, A\, n)\, \Sigma & = & \Sigma\, n = (\texttt{Internal}, A) \vee \Sigma\, n = (\texttt{Output}\, \texttt{true}, A)
\end{array}
$$

Note that, as with the definition of `read` and `write` in the previous section, the type contained in a cell may differ from the type of the resource. Given a resource $r : \mathit{Ref}\ A$ and an abstract memory Σ, we denote $r \vdash \Sigma$ if the type of the tag associated with r in Σ is A. The abstraction relation defined by γ ensures that types remain consistent in a concrete memory, provided they are consistent in an abstraction of it. The following lemma establishes that resource accesses preserve abstractions.

Lemma 6. *Let r be a resource of type $\mathit{Ref}\ A$, σ be a memory and let Σ be an abstract memory such that $r \vdash \Sigma$:*

- *if $\sigma \in \gamma\, \Sigma$ and $\texttt{read}^\dagger\, r\, \Sigma = \Sigma'$, then there exists a value a of type A and a memory σ' such that $\texttt{read}\, r\, \sigma = (a, \sigma')$, $\sigma' \in \gamma\, \Sigma'$ and $r \vdash \Sigma'$. Moreover, for all resource $r' \neq r$, $\Sigma\, r' = \Sigma'\, r'$.*
- *if $\sigma \in \gamma\, \Sigma$ and $\texttt{write}^\dagger\, r\, \Sigma = \Sigma'$, then, for any value a of type A, there exists a memory σ' such that $\texttt{write}\, r\, \sigma\, a = \sigma'$, $\sigma' \in \gamma\, \Sigma'$ and $r \vdash \Sigma'$. Moreover, for all resource $r' \neq r$, $\Sigma\, r' = \Sigma'\, r'$.*

Proof. Immediate by definition of `read`, `write`, $\texttt{read}^\dagger$ and $\texttt{write}^\dagger$. □

5.4.2 Abstract semantics

The abstract semantics of a term is defined as a partial function over abstract memories. For terms of the form $\texttt{Arr}\ f$, the abstract memory remains unchanged. For terms of the form $\texttt{Get}\ r$ and $\texttt{Set}\ r$, the abstract semantics updates the state of memory cells using $\texttt{read}^\dagger$ and $\texttt{write}^\dagger$ operations. For terms of the form `First` or `Comp`, changes in the abstract memory propagate.

$$
\begin{array}{lcll}
\texttt{step}^\dagger & : & & \mathit{RSF} \to \mathit{Memory}^\dagger \rightharpoonup \mathit{Memory}^\dagger \\
\texttt{step}^\dagger\, (\texttt{Arr}\, f)\, \Sigma & = & \Sigma & \\
\texttt{step}^\dagger\, (\texttt{First}\, \mathit{rsf})\, \Sigma & = & \Sigma' & \text{if } \texttt{step}^\dagger\, \mathit{rsf}\, \Sigma = \Sigma' \\
\texttt{step}^\dagger\, (\texttt{Comp}\, (\mathit{rsf}_1, \mathit{rsf}_2))\, \Sigma & = & \Sigma' & \text{if } \texttt{step}^\dagger\, \mathit{rsf}_1\, \Sigma = \Sigma'' \wedge \texttt{step}^\dagger\, \mathit{rsf}_2\, \Sigma'' = \Sigma' \\
\texttt{step}^\dagger\, (\texttt{Get}\, r)\, \Sigma & = & \Sigma' & \text{if } \texttt{readable}^\dagger\, r\, \Sigma \wedge \texttt{read}^\dagger\, r\, \Sigma = \Sigma' \\
\texttt{step}^\dagger\, (\texttt{Set}\, r)\, \Sigma & = & \Sigma' & \text{if } \texttt{writable}^\dagger\, r\, \Sigma \wedge \texttt{write}^\dagger\, r\, \Sigma = \Sigma'
\end{array}
$$

A program p is well-typed if its abstract semantics is defined over an initial abstract memory given by $\texttt{init}^\dagger\, p$, where all inputs are readable ($\texttt{Input}\, \texttt{true}$) and all outputs are writable ($\texttt{Output}\, \texttt{true}$). Furthermore, we impose an additional condition that in the resulting abstract memory, all inputs must have the tag $\texttt{Input}\, \texttt{false}$ and all outputs must have the tag $\texttt{Output}\, \texttt{false}$. The condition on output resources is mandatory to ensure that the program produces outputs at every step. The condition over input resources is optional, but useful to ensure that the program consumes all its inputs. Below we define the $\texttt{init}^\dagger$ functions and the well-typedness condition.

$$
\begin{array}{lcl}
\texttt{init}^\dagger & : & \mathit{Prog} \to \mathit{Memory}^\dagger \\
\texttt{init}^\dagger\, p\, n & = & \begin{cases}
(\texttt{Input}\, \texttt{true}, \tau) & \text{if } n < k_{in}\, p \text{ and } \tau = (\texttt{inputs}\, p)_n \\
(\texttt{Output}\, \texttt{false}, \tau) & \text{if } k_{in}\, p \leq n < k_{in}\, p + k_{out}\, p \\
 & \text{and } \tau = (\texttt{outputs}\, p)_{n - k_{in}\, p} \\
(\texttt{Internal}, \tau) & \text{if } k_{in}\, p + k_{out}\, p \leq n < k_{in}\, p + k_{out}\, p + k\, p \\
 & \text{and } (v : \tau) = (\texttt{internals}\, p)_{n - (k_{in}\, p + k_{out}\, p)}, \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{array}
$$

Definition 1. *A program p is well-typed if* $\texttt{step}^\dagger\,(\texttt{program}\,p)\,(\texttt{init}^\dagger\,p)$ *is defined and the following conditions hold:*

- *for all* $n < k_{in}$ *we have* $(\texttt{step}^\dagger\,(\texttt{program}\,p)\,(\texttt{init}^\dagger\,p))\,n = (\texttt{Input}\,\texttt{false}, \cdot)$
- *for all* $k_{in}\,p \leq n < k_{in}\,p + k_{out}\,p$ *we have* $(\texttt{step}^\dagger\,(\texttt{program}\,p)\,(\texttt{init}^\dagger\,p))\,n = (\texttt{Output}\,\texttt{false}, \cdot)$

We conclude this section with the following results, which establish that well-typed programs are reactive, i.e. they produce outputs for all possible inputs.

Theorem 4 (Correctness)**.** *Let p be a well-typed program and, i be a list compatible with* $\texttt{inputs}\,p$*, and σ be a memory such that* $\texttt{pull}\;p\,\sigma\,i \in \gamma\,(\texttt{init}^\dagger\,p)$*. Then, there exists σ' such that:*

- $\texttt{step}\,(\texttt{program}\,p)\,\texttt{tt}\,(\texttt{pull}\,p\,\sigma\,i) = (\texttt{tt}, \sigma')$
- $\texttt{push}\;p\,\sigma' : (\alpha\,(\texttt{outputs}\,p))$
- $\texttt{pull}\;p\,\sigma'\,i' \in \gamma\,(\texttt{init}^\dagger\,p)$*, for all i' of type* $(\alpha\,(\texttt{inputs}\,p))$

Proof. First, we prove the following result: for all sf, σ, a and Σ, such that $\sigma \in \gamma\,\Sigma$ and $\texttt{step}^\dagger\;sf\;\Sigma = \Sigma'$ for some Σ', there exists σ' such that $sf\;a\,\sigma = (b, \sigma')$ and $\sigma' \in \gamma\,\Sigma'$. The proof proceeds by induction on the structure of sf. Second we use it for our current goal. By lemma 6, the abstract semantics preserves the classification of memory cells as internals, input or output. In particular, σ' remains compatible with $\texttt{init}^\dagger\,p$ for all internal resources. Moreover, it is straightforward to verify that for all resource r of the program we have $r \vdash \texttt{init}^\dagger\,p$. From this, it follows that $\texttt{pull}\,\sigma'\,'i \in \gamma\,(\texttt{init}^\dagger\,p)$ for all i' compatible with $\texttt{inputs}\,p$. The final result follows by applying these properties to the body of p. □

Corollary 1 (Reactivity)**.** *Let p be a well-typed program. Then for all input streams $is : Stream\,(\alpha\,(\texttt{inputs}\,p))$, there exists an output stream $os : Stream\,(\alpha\,(\texttt{outputs}\,p))$ such that* $\texttt{run}\,p\,(\texttt{init}\,p)\,is = os$*.*

Proof. By bisimulation on streams, using theorem 4. □

For the remainder of this paper, we consider only well-typed programs applied to inputs of the correct type. Consequently, based on results of this section, we can assume that the stepwise semantics and, by extension the semantics over streams, are total functions.

6 Equivalence

In this section we state and prove the main result of this paper: the existence of a normal form for MOLHOLES terms, where all resource accesses occur at the top level, and all internal resources are read and written at most once. This first result is established in section 6.1. Furthermore, we show that these normal forms can be transformed into semantically equivalent YAMPACORE normal forms. This second result is established in section 6.2. We conjecture that similar results hold for WORMHOLES terms, but to the best of our knowledge, this has never been stated nor proven. Thus, our work extends previous work not only by relaxing constraints on resource usage but also by establishing the existence of normal forms and proving their equivalence to YAMPACORE terms. Another result, not detailed here for brevity, is that YAMPACORE normal forms can also be transformed into MOLHOLES normal forms. Together, these results, all of which rely on the equational theories presented in previous sections and on bisimulation proofs, establish that MOLHOLES remains compatible with existing functional paradigms.

6.1 Normal form

A term t may contain a set of resources which can be separated between the read ones and the written ones. We define $\texttt{refs}_{get}$, respectively $\texttt{refs}_{set}$, a function that takes a term t and returns an ordered list of resources used for read, respectively for write, without multiple occurrences. With these functions declared, we state that a term in normal form has the following structure

$$\texttt{Get}\,r'_0 \ggg \ldots \ggg \texttt{Get}\,r'_{n_g-1} \ggg \texttt{Arr}\,f \ggg \texttt{Set}\,r'_{n_s-1} \ggg \ldots \ggg \texttt{Set}\,r'_0$$

with function f from the host language, and $n_g = |\texttt{refs}_{get}(t)|$ and $n_s = |\texttt{refs}_{set}(t)|$. For readability, we use the infix operator $(\ggg)$ instead of `Comp`, which is left associative. Knowing that, resource identifiers for inputs are lower than ones for internals, and internals ones are lower than outputs ones, we can add that input and output operations are at the outermost of the term.

Example 3. *For instance, defining* $add = \lambda(x, y).x + y$*, we expect to prove that the following holds:*

$$\texttt{Arr}\,(dup) >\!\!>\!\!> \texttt{Set}\,r' >\!\!>\!\!> \texttt{Get}\,r >\!\!>\!\!> \texttt{Arr}\,add \equiv \texttt{Get}\,r >\!\!>\!\!> \texttt{Arr}\,f >\!\!>\!\!> \texttt{Set}\,r'$$

where $f = \lambda(x, y).\texttt{let}\ x' = dup\,x\ \texttt{in}\ \texttt{let}\ (w, z) = perm\ (x', y)\ \texttt{in}\ (add\ w, z)$.

$$seq_l(\texttt{GS}, l, t) = \begin{cases} t & \text{si } l = [] \\ \texttt{Comp}\,(seq'_l(\texttt{GS}, l), t) & \text{sinon} \end{cases}$$

$$seq'_l(\texttt{GS}, l) = \begin{cases} \texttt{Arr id} & \text{si } l = [] \\ \texttt{Comp}\,(seq'_l(\texttt{GS}, l'), \texttt{Get}\,r) & \text{si } l = l' +\!\!+ [r] \text{ et } \texttt{GS} = \texttt{G} \\ \texttt{Comp}\,(seq'_l(\texttt{GS}, l'), \texttt{Set}\,r) & \text{si } l = l' +\!\!+ [r] \text{ et } \texttt{GS} = \texttt{S} \end{cases}$$

$$seq_r(\texttt{GS}, l, t) = \begin{cases} t & \text{si } l = [] \\ \texttt{Comp}\,(seq_r(\texttt{GS}, l', t), \texttt{Get}\,r) & \text{sinon, où } l = r :: l' \text{ et } \texttt{GS} = \texttt{G} \\ \texttt{Comp}\,(seq_r(\texttt{GS}, l', t), \texttt{Set}\,r) & \text{sinon, où } l = r :: l' \text{ et } \texttt{GS} = \texttt{S} \end{cases}$$

$$stack(n, t) = \begin{cases} t & \text{si } n = 0 \\ \texttt{First}\,(stack(m, t)) & \text{sinon, où } n = m + 1 \end{cases}$$

$$perm_l(n, t) = \begin{cases} t & \text{si } n = 0 \\ \texttt{Comp}\,(perm'_l(0, m), t) & \text{sinon, où } n = m + 1 \end{cases}$$

$$perm'_l(n, m) = \begin{cases} stack(m, \texttt{Arr}\,perm) & \text{si } n \geq m \\ \texttt{Comp}\,(perm'_l(n + 1, m), stack(n, \texttt{Arr}\,perm)) & \text{sinon} \end{cases}$$

$$perm_r(n, t) = \begin{cases} t & \text{si } n = 0 \\ \texttt{Comp}\,(perm_r(m, t), stack(m, \texttt{Arr}\,perm)) & \text{sinon,} \\ & \text{où } n = m + 1 \end{cases}$$

Figure 7: Concise notation for parameterized terms

The rest of this section is dedicated to proving the normal form theorem. Before proceeding with the proof, we introduce several intermediary results and define functions that enable a concise representation of terms involving sequences of `First` constructs, resources accesses, and permutation arrows. These functions are detailed in fig. 7. They satisfy the set of equations 12 given in appendix.

A term is said to effect-free if it does not contain any `Get` or `Set` operation. lemma 7 states that any effect-free term can be contracted to a simple arrow.

Lemma 7 (Contraction)**.** *For any effect-free term* t*, there exists a function* f *in the host language such that* $t \equiv \texttt{Arr}\,f$*.*

lemma 8 states that we can swap sequences of read and write operations, while lemma 9 states that we can merge two sequences of read operations or two sequences of write operations. For brevey, we omit the proofs of these lemmas. In each case, the proof proceeds by structural induction on lists, except for lemma 7 which is proven by structural induction on the term.

Lemma 8 (Swap)**.** *For any function* g *and any two lists of resources* rs_1 *and* rs_2*, there exists a function* f *such that*

$$seq_r(\texttt{G}, rs_1, seq_l(\texttt{S}, rs_2, \texttt{Arr}\,g)) \equiv seq_r(\texttt{S}, rs'_2, seq_l(\texttt{G}, rs'_1, \texttt{Arr}\,f))$$

with $rs'_1 = rs_1 \backslash rs_2$ *and* $rs'_2 = rs_2 \backslash rs_1$*.*

Lemma 9 (Merge)**.** *For any function* g *and two sorted list of resources* rs_1 *and* rs_2*, there exists a function* f *such that*

$$seq_l(\texttt{G}, rs_1, seq_l(\texttt{G}, rs_2, \texttt{Arr}\,g)) \equiv seq_l(\texttt{G}, rs, \texttt{Arr}\,f)$$
$$seq_r(\texttt{S}, rs_1, seq_r(\texttt{S}, rs_2, \texttt{Arr}\,g)) \equiv seq_r(\texttt{S}, rs, \texttt{Arr}\,f)$$

with rs *the sorted concatenation of* rs_1 *and* rs_2*.*

We conclude this section by stating and proving the normal form theorem.

Theorem 5 (Normal form)**.** *For any term t, there exists a function f of the host language such that $t \equiv seq_r^{\mathtt{S}}(\mathtt{refs}_{set}(t), seq_l^{\mathtt{G}}(\mathtt{refs}_{get}(t), \mathtt{Arr}\, f))$, i.e.*

$$t \equiv \mathtt{Get}\, r_0 \ggg \ldots \ggg \mathtt{Get}\, r_{n_g-1} \ggg \mathtt{Arr}\, f \ggg \mathtt{Set}\, r'_0 \ggg \ldots \ggg \mathtt{Set}\, r'_{n_s-1}$$

where $n_g = |\mathtt{refs}_{get}(t)|$ and $n_s = |\mathtt{refs}_{set}(t)|$.

Proof. The proof is by structural induction on the term t.

- $t = \mathtt{Arr}\, g$: on a $\mathtt{refs}_{get}(t) = \mathtt{refs}_{set}(t) = \{\}$, which simplifies the goal as follows $\exists f, \mathtt{Arr}\, g \equiv \mathtt{Arr}\, f$. Consequently, this case is straightforward by defining f by g.
- $t = \mathtt{First}\, t'$: by induction hypothesis, we know that :

  $$\exists f, t' \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), \mathtt{Arr}\, f))$$

  We pose f and replace t' in $\mathtt{First}\, t'$. Now we have to move the $\mathtt{First}$ under both sequences.

  $$\begin{aligned}
  & \mathtt{First}\, (seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), \mathtt{Arr}\, f))) \\
  \equiv\ & seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), \\
  & \quad perm_r(n_s, \mathtt{First}\, (seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), \mathtt{Arr}\, f)))) && \text{(éq. } 12n\text{)} \\
  \equiv\ & seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), \\
  & \quad perm_r(n_s, seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), perm_r(n_g, \mathtt{First}\, (\mathtt{Arr}\, f))))) && \text{(éq. } 12m\text{)} \\
  \equiv\ & seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), \\
  & \quad seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), perm_r(n_s, perm_r(n_g, \mathtt{First}\, (\mathtt{Arr}\, f))))) && \text{(éq. } 12e\text{)}
  \end{aligned}$$

  where $n_g = |\mathtt{refs}_{get}(t)|$ and $n_s = |\mathtt{refs}_{set}(t)|$. By lemma 7, we define $\mathtt{Arr}\, g \equiv perm_r(n_s, perm_r(n_g, \mathtt{First}\, (\mathtt{Arr}\, f)))$. Consequently, there exists a function g such that

  $$\mathtt{First}\, t \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), \mathtt{Arr}\, g))$$

- $t = \mathtt{Get}\, r$: we define f as the identity function $\mathtt{id}$. By eq. (5b) we can rewrite the term as follows

  $$\mathtt{Get}\, r \equiv seq_r(\mathtt{S}, [], seq_l(\mathtt{G}, [r], \mathtt{Arr}\, \mathtt{id})) \equiv \mathtt{Comp}\, (\mathtt{Get}\, r)\, (\mathtt{Arr}\, \mathtt{id}) \equiv (\mathtt{Get}\, r)$$

- $t = \mathtt{Set}\, r$: we define f as the identity function $\mathtt{id}$. By eq. (5a) we know that

  $$\mathtt{Set}\, r \equiv seq_r(\mathtt{S}, [r], seq_l(\mathtt{G}, [], \mathtt{Arr}\, \mathtt{id})) \equiv \mathtt{Comp}\, (\mathtt{Arr}\, \mathtt{id})\, (\mathtt{Set}\, r) \equiv (\mathtt{Set}\, r)$$

- $t = \mathtt{Comp}\, t_1\, t_2$: By induction hypothesis, we know that

  $$\begin{aligned}
  & \exists f_1, t_1 \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t_1), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t_1), \mathtt{Arr}\, f_1)) \\
  & \exists f_2, t_2 \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t_2), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t_2), \mathtt{Arr}\, f_2))
  \end{aligned}$$

  We pose f_1, f_2 and replace t_1 and t_2 in t. Moreover, we define $r_{g_1} = \mathtt{refs}_{get}(t_1)$, $r_{s_1} = \mathtt{refs}_{set}(t_1)$, $r_{g_2} = \mathtt{refs}_{get}(t_2)$ and $r_{s_2} = \mathtt{refs}_{set}(t_2)$.

  $$t = \mathtt{Comp}\, (seq_r(\mathtt{S}, r_{s_1}, seq_l(\mathtt{G}, r_{g_1}, \mathtt{Arr}\, f_1)), seq_r(\mathtt{S}, r_{s_2}, seq_l(\mathtt{G}, r_{g_2}, \mathtt{Arr}\, f_2)))$$

  Thanks to the composition associativity, i.e. eq. (5c), and the composition neutral, i.e. eqs. (5a) and (5b), we can rewrite t as follows

  $$\begin{aligned}
  t \equiv \mathtt{Comp}\, (\mathtt{Comp}\, & (seq_l(\mathtt{G}, r_{g_1}, \mathtt{Arr}\, f_1), \\
  & \quad seq_r(\mathtt{S}, r_{s_1}, seq_l(\mathtt{G}, r_{g_2}, \mathtt{Arr}\, \mathtt{id}))), \\
  & seq_r(\mathtt{S}, r_{s_2}, \mathtt{Arr}\, f_2))
  \end{aligned}$$

  By lemma 8, there exists a function f such that

  $$\begin{aligned}
  t \equiv \mathtt{Comp}\, (\mathtt{Comp}\, & (seq_l(\mathtt{G}, r_{g_1}, \mathtt{Arr}\, f_1), \\
  & \quad seq_l(\mathtt{G}, r'_{g_2}, seq_r(\mathtt{S}, r'_{s_1}, \mathtt{Arr}\, f))), \\
  & seq_r(\mathtt{S}, r_{s_2}, \mathtt{Arr}\, f_2))
  \end{aligned}$$

with $r'_{s_1} = r_{s_1} \backslash r_{g_2}$ and $r'_{g_2} = r_{g_2} \backslash r_{s_1}$. By eqs. (12g) to (12i), which are the associativity and neutral, we can rewrite t as follows

$$t \equiv \texttt{Comp}\,(seq_l(\texttt{G}, r_{g_1}, seq_l(\texttt{G}, r'_{g_2}, stack(|r'_{g_2}|, \texttt{Arr}\ f_1))),\\ seq_r(\texttt{S}, r_{s_2}, seq_r(\texttt{S}, r'_{s_1}, stack(|r'_{s_1}|, \texttt{Arr}\ f_2))))$$

By lemma 7, we define f'_1, f'_2 such that $\texttt{Arr}\ f'_1 \equiv stack(|r'_{g_2}|, \texttt{Arr}\ f_1)$ and $\texttt{Arr}\ f'_2 \equiv stack(|r'_{s_1}|, \texttt{Arr}\ f_2)$. Finally, we apply on both sub-term of the composition the lemma 9 which give us two functions f''_1 and f''_2 as well as two lists of resources r_g and r_s. We can rewrite t as follows

$$t \equiv seq_r(\texttt{S}, r_s, seq_l(\texttt{G}, r_g, \texttt{Comp}\,(\texttt{Arr}\ f''_1, \texttt{Arr}\ f''_2)))$$

The eq. (5e) allows us to hide the composition below an `Arr` term. Consequently, there exists $f = f''_2 \circ f''_1$ which satisfies the equivalence.

□

6.2 Transformation into YAMPA

From a program p obtained through the previous transformation, it is possible to construct an equivalent version where all resources of the same kind are merged into a single resource, and all read (resp. write) accesses to resources of the same kind are merged into a single read (resp. write) access. For lack of space, we do not detail this transformation here. Instead, we assume that the program obtained from the previous section takes the following form:

$$\begin{aligned} p' = \quad \{ \quad & \texttt{inputs} = [((\tau_1 \times \tau_2) \times \ldots) \times \tau_n]; \qquad \texttt{outputs} = [((\tau'_1 \times \tau'_2) \times \ldots) \times \tau'_l];\\ & \texttt{internals} = [(((v_{n+l+1} : \tau_{n+l+1}, v_{n+l+2} : \tau_{n+l+2}), \ldots), v_{n+l+m} : \tau_{n+l+m})];\\ & \texttt{program} = \texttt{Comp}\,(\texttt{Get}\ r_{in})\,(\texttt{Comp}\,(\texttt{Get}\ r)\,(\texttt{Comp}\,(\texttt{Arr}\ f)\,(\texttt{Comp}\,(\texttt{Set}\ r)\,(\texttt{Set}\ r_{out}))))\} \end{aligned}$$

where $n = k_{in}\,p$, $m = k\,p$ and $l = k_{out}\,p$. Types and values results from collapsing their counterpart in p and, r_{in},r and r_{out} are the single input, internal and output resources, respectively. The function f has type $((Unit \times \texttt{inputs}\,p') \times \texttt{internals}\,p') \to ((Unit \times \texttt{outputs}\,p') \times \texttt{internals}\,p')$.

Given a program p of the form described above, we define

$$C_p = \{\sigma \mid \forall i : (\alpha(\texttt{inputs}\,p)), \texttt{pull}\,p\,\sigma\,i \in \gamma\,(\texttt{init}^\dagger\,p)\}$$

Up to permutation of the parameters of $\texttt{step}\,(\texttt{program}\,p)$, the pair $(C_p\ \texttt{step}\,(\texttt{program}\,p))$ forms a coalgebra for the functor F_{IO}, where $[I] = \texttt{inputs}\,p$ and $[O] = \texttt{outputs}\,p$. Thus, we can define a bisimulation between $(C_p, \texttt{step}\,(\texttt{program}\,p))$ and $(sf\ I\ O,\ id)$. A relation R is a bisimulation if for all $\sigma \in C_p$, $a : I$, $b : O$ and $sf : sf\ I\ O$ such that $R\,\sigma\,sf$,

$$\begin{aligned} &\text{if } \texttt{step}\,(\texttt{program}\,p)\,\texttt{tt}\,(\texttt{pull}\,p\,\sigma\,[a]) = (\texttt{tt}, \sigma') \text{ then there exists } sf' \text{ such that}\\ &\quad sf\ a = (b, sf'), \text{where } [b] = \texttt{push}\,p\,\sigma',\ \text{and } R\,\sigma'\,sf' \end{aligned}$$

The transformation of a program p into YAMPA is expressed as $translate\,p\,(\texttt{init}\,p)$ where, f being the function appearing in the definition of p, and v the value that aggregates the initial values of the internal resources, the function $translate$ is defined as:

$$translate\,p\,\sigma = \texttt{Loop}\,v\,(\texttt{Arr}\ f)$$

With this definition, the initial memory of the program p is bisimilar to the semantics of its translation (proposition 1). As a corollary, we conclude that p and its transformation denote the same stream function (corollary 2).

Proposition 1. *For all well-typed program p, the following holds:*

$$\texttt{init}\,p \sim \texttt{step}\,(translate\,p\,(\texttt{init}\,p))$$

where $\sim$ denotes bisimilarity between $(C_p, \texttt{step}\,(\texttt{program}\,p))$ and $(sf\ I\ O, id)$.

Proof. First, prove that if $\texttt{step}\,(translate\,p\,\sigma)\,a = (b, sf)$ then $sf = translate\,p\,\sigma'$ for some appropriate σ'. The key observation is that both $\lambda a.\texttt{step}\,(\texttt{program}\,p)\,\texttt{tt}\,(pull\,p\,\sigma\,a)$ and $translate\,p\,\sigma$ apply the same function f to values occurring in the inputs on one side, and values of internal resources and loop states on the other side. Thus, the resulting values are the same of both side, up to a shift between inputs on one side and internal resources and loop states on the other side. Next, prove that $R = \{(\sigma, \texttt{step}\,(translate\,p\,\sigma)) \mid \sigma \in C_p\}$ is a bisimulation. □

Corollary 2. *For any well-typed program p, p and $translate\,p\,(\texttt{init}\,p)$ denote the same stream function, i.e. $\texttt{run}\,p\,(\texttt{init}\,p)\,s \sim \texttt{run}\,(translate\,p\,(\texttt{init}\,p))\,s$ where $\sim$ denotes bisimilarity between streams.*

Proof. This follows by applying proposition 1, along with the definitions of the two `run` functions and bisimulation, and the bisimulation principle for streams. □

As mentioned earlier, one can also show that the transformation between normal forms is reversible, thereby establishing that MOLHOLES and YAMPACORE are equivalent in terms of expressivity.

7 Conclusion

We have introduced MOLHOLES, a programming language that extends WORMHOLES by relaxing constraints on resource usage. Specifically, MOLHOLES permits multiple reads and writes to internal resources while still enforcing the single-use policy for input and output resources. A type system ensures correct resource usage.

We have demonstrated that well-typed programs can be transformed into equivalent versions where all resources are accessed at most once and at the top level. These transformations are justified using the equational theories of categories, functors, monads, and arrows, along with additional equalities specific to the state monad used in the language's semantics.

Building on this transformation, we have shown that MOLHOLES programs can be translated into equivalent resource-free YAMPACORE programs. Since the semantics of YAMPACORE arrows are defined using corecursive functions, this equivalence is established via bisimulation proof techniques. Together, these results confirm that our approach preserves the functional nature of YAMPA while offering a more flexible resource management system than WORMHOLES.

This work was initially motivated by the development of a reactive programming library (currently in progress) for the OCAML language [23], integrating YAMPA's programming model with OCAML's impure nature.

For future work, we plan to formalize our results using the COQ proof assistant [22] and extract certified programs from the formalization. Our long-term goal is to verify program correctness against behavioral specifications, potentially expressed in temporal logic, enabling the generation of certified reactive programs. We expect the semantics domain of MOLHOLES to be expressive enough to define the semantics of similar languages, i.e., languages that combine the dataflow approach with effectful computations.

Another research direction involves extending MOLHOLES with additional constructs. The combinators discussed in this paper cover only a restricted subset of signal functions, which simplifies bisimulation due to the limited update behavior in computation steps. We aim to investigate YAMPA combinators beyond this restriction and analyze their impact on bisimulation proofs. Additionally, we plan to explore language extensions that enhance expressivity, including other computational effects, such as exception handling, to improve compatibility with OCAML. In particular, the interaction between resources and OCAML's references is of interest.

References

[1] Robert Atkey. What is a categorical model of arrows? *Electronic Notes in Theoretical Computer Science*, 229(5):19–37, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). doi:10.1016/j.entcs.2011.02.014.

[2] Patrick Bahr. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *J. Funct. Program.*, 32:e15, 2022. doi:10.1017/S0956796822000132.

[3] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991. doi:10.1016/0167-6423(91)90001-E.

[4] Gérard Berry. *The foundations of Esterel*. MIT Press, Cambridge, MA, USA, 2000. doi:10.7551/mitpress/5641.003.0021.

[5] Frédéric Boussinot. Reactive C: an extension of C to program reactive systems. *Software-Practice and Experience (SPE)*, 21(4):401–428, 1991. doi:10.1002/spe.4380210406.

[6] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *Transactions of Software Engineering (TSE)*, 22(4):256–266, 1996. doi:10.1109/32.491649.

[7] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. *Electron. Notes Theor. Comput. Sci.*, 11:1–21, 1998. doi:10.1016/S1571-0661(04)00050-7.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*, pages 263–273, Amsterdam, The Netherlands, 1997. ACM. doi:10.1145/258948.258973.

[9] Conal M. Elliott. Push-pull functional reactive programming. In *Symposium on Haskell*, pages 25–36, Edinburgh, Scotland, UK, 2009. ACM. doi:10.1145/1596638.1596643.

[10] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proc. Inst. Electr. Electron. Eng.*, 79(9):1305–1320, 1991. doi:10.1109/5.97300.

[11] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. doi:10.1007/978-3-642-82453-1_17.

[12] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming (AFP)*, volume 2638 of *LNCS*, pages 159–187, Oxford, UK, 2002. Springer. doi:10.1007/978-3-540-44833-4_6.

[13] John Hughes. Generalizing monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000. doi:10.1016/S0167-6423(99)00023-4.

[14] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

[15] Louis Mandel and Marc Pouzet. ReactiveML: a reactive extension to ML. In *Principles and Practice of Declarative Programming (PPDP)*, pages 82–93, Lisbon, Portugal, 2005. ACM. doi:10.1145/1069774.1069782.

[16] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science. doi:10.1016/0890-5401(91)90052-4.

[17] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 33–44, New York, NY, USA, 2016. ACM. doi:10.1145/2976002.2976010.

[18] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: www.lri.fr/∼pouzet/lucid-synchrone.

[19] Davide Sangiorgi and Jan Rutten, editors. *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011.

[20] Abhiroop Sarkar and Mary Sheeran. Hailstorm: A statically-typed, purely functional language for IoT applications. *Principles and Practice of Declarative Programming (PPDP)*, 2020. doi:10.1145/3414080.3414092.

[21] David I. Spivak. *Category Theory for the Sciences*. The MIT Press, 2014.

[22] The Coq Development Team. The Coq Proof Assistant. Distribution available at: http://coq.inria.fr.

[23] The OCaml Development Team. The OCaml Programming Language. Distribution available at: https://ocaml.org.

[24] Atze van der Ploeg and Koen Claessen. Practical principled FRP: forget the past, change the future, frpnow! In *International Conference on Functional Programming (ICFP)*, pages 302–314, Vancouver, BC, Canada, 2015. ACM. doi:10.1145/2784731.2784752.

[25] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/91556.91592.

[26] Daniel Winograd-Cort and Paul Hudak. Wormholes: Introducing effects to FRP. *SIGPLAN Not.*, 47(12):91–104, 2012. doi:10.1145/2430532.2364519.

[27] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. Virtualizing real-world objects in FRP. In *Practical Aspects of Declarative Languages (PADL)*, volume 7149 of *LNCS*, pages 227–241. Springer, 2012. doi:10.1007/978-3-642-27694-1_17.

[28] Dana N. Xu and Siau-Cheng Khoo. Compiling real time functional reactive programming. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, page 83–93, New York, NY, USA, 2002. ACM. doi:10.1145/568173.568183.

$$seq_l(\mathtt{G}, rs +\!\!+ [r], t) \equiv seq_l(\mathtt{G}, rs, \mathtt{Comp}\,(\mathtt{Get}\,r, t)) \tag{12a}$$
$$seq_l(\mathtt{S}, rs +\!\!+ [r], t) \equiv seq_l(\mathtt{S}, rs, \mathtt{Comp}\,(\mathtt{Set}\,r, t)) \tag{12b}$$
$$seq_l(\mathtt{GS}, rs, seq_l(\mathtt{GS}, rs', t)) \equiv seq_l(\mathtt{GS}, rs +\!\!+ rs', t) \tag{12c}$$
$$seq_r(\mathtt{GS}, rs, seq_r(\mathtt{GS}, rs', t)) \equiv seq_r(\mathtt{GS}, rs +\!\!+ rs', t) \tag{12d}$$
$$seq_l(\mathtt{GS}, rs, perm_r(n, t)) \equiv perm_r(n, seq_l(\mathtt{GS}, rs, t)) \tag{12e}$$
$$seq_r(\mathtt{GS}, rs, perm_l(n, t)) \equiv perm_l(n, seq_r(\mathtt{GS}, rs, t)) \tag{12f}$$

$$seq_r(\mathtt{G}, rs, t') \equiv seq_l(\mathtt{G}, rs, stack(|rs|, t')) \tag{12g}$$
$$seq_l(\mathtt{S}, rs, t') \equiv seq_r(\mathtt{S}, rs, stack(|rs|, t')) \tag{12h}$$
$$seq_l(\mathtt{G}, rs, \mathtt{Comp}\,(stack(|rs|, t'), t)) \equiv \mathtt{Comp}\,(t', seq_l(\mathtt{G}, rs, t)) \tag{12i}$$
$$seq_r(\mathtt{S}, rs, \mathtt{Comp}\,(t, stack(|rs|, t'))) \equiv \mathtt{Comp}\,(seq_r(\mathtt{S}, rs, t), t') \tag{12j}$$

$$perm_l(n, \mathtt{Comp}\,(\mathtt{Get}\,r, \mathtt{Comp}\,(\mathtt{Arr}\,perm, t))) \equiv \mathtt{Comp}\,(\mathtt{Get}\,r, perm_l(n+1, t)) \tag{12k}$$
$$perm_r(n, \mathtt{Comp}\,(t, \mathtt{Comp}\,(\mathtt{Arr}\,perm, \mathtt{Set}\,r))) \equiv \mathtt{Comp}\,(perm_r(n+1, t), \mathtt{Set}\,r) \tag{12l}$$
$$\mathtt{First}\,(seq_l(\mathtt{G}, rs, t)) \equiv seq_l(\mathtt{G}, rs, perm_l(|rs|, \mathtt{First}\,t)) \tag{12m}$$
$$\mathtt{First}\,(seq_r(\mathtt{S}, rs, t)) \equiv seq_r(\mathtt{S}, rs, perm_r(|rs|, \mathtt{First}\,t)) \tag{12n}$$
$$seq_l(\mathtt{G}, rs, \mathtt{Comp}\,(perm_r(|rs|, \mathtt{Get}\,r), t)) \equiv \mathtt{Comp}\,(\mathtt{Get}\,r, seq_l(\mathtt{G}, rs, t)) \tag{12o}$$
$$seq_r(\mathtt{S}, rs, \mathtt{Comp}\,t, (perm_l(|rs|, \mathtt{Set}\,r))) \equiv \mathtt{Comp}\,(seq_r(\mathtt{G}, rs, t), \mathtt{Set}\,r) \tag{12p}$$
$$\mathtt{Comp}\,(\mathtt{Get}\,r, perm_r(n, t)) \equiv perm_r(n, \mathtt{Comp}\,(\mathtt{Get}\,r, t)) \tag{12q}$$
$$\mathtt{Comp}\,(\mathtt{Set}\,r, perm_r(n, t)) \equiv perm_r(n, \mathtt{Comp}\,(\mathtt{Set}\,r, t)) \tag{12r}$$