A new paradigm for component-based development

Johan G. Granström

***Abstract*—Hancock and Setzer [10] describe how Haskell's monolithic IO monad can be decomposed into *worlds* when working in a dependently typed language, like Martin-Löf's type theory [15].**

This paper introduces the notion of *world map* and shows that worlds and world maps form a category with arbitrary products. The construction of the category is carried out entirely in type theory and directly implementable in dependently typed programming languages.

If we let the notion of world replace the standard notion of *interface*, and the notion of world map replace the standard notion of *component*, we get a rigorous paradigm for component-based development. This new paradigm is investigated and several applications are given. For example, the problem of session state is given a very clean solution (p. 8).

***Index Terms*—functional constructs, components, functional programming, semantics of programming languages**

INTRODUCTION

HISTORICALLY, IO has been a weak spot for pure (side effect free) functional programming languages. The seminal functional programming language FP [3] completely lacked IO facilities; early versions of Haskell used stream based IO [12]; and there is still no consensus about how to perform IO in Martin-Löf's type theory [15].

This paper builds on the work of Hancock and Setzer [10] on *worlds* and *interactive programs*. Their work builds on Moggi's work [17] on *monads* in functional programming and is presented in the language of Martin-Löf's type theory [15].

Today, monads are at the core of Haskell's IO system, and any computation with side effects has to be wrapped in a monad. Haskell monads have been very successful, but monads' unwillingness to compose has been an obstacle.

> [14]: Soon, however, it became clear that despite the undoubted value of monads from both the semantic and programming perspectives, composing monads would prove to be a significant challenge. Put briefly, monads just do not seem to compose in any general manner.

Worlds were originally an attempt to solve these problems, i.e., to provide compositional IO and state manipulation based on dependent type theory. World maps, introduced in this paper, takes this idea one step further and suggests a whole new component-based paradigm based on a category of worlds and world maps.

My grand vision is that the component-based paradigm eventually will replace the object-oriented paradigm as the leading paradigm of software development. Every long journey starts with one step...

Email: georg.granstrom@gmail.com. Current affiliation: Google Zürich Gmbh, Brandschenkestrasse 110, 8002 Zürich.

Main results

A world w is a dependently typed pair

$$(x : |w|,\ w@x),$$

where $|w|$ is a set and $w@x$ is a set, for any $x : |w|$.

For a world w, there is a monad whose type constructor maps a set A to the set

$$w \Rightarrow A$$

of interactive programs over w (Thm. 2). The intuition behind the notation $w \Rightarrow A$ is that an object of this type maps a realization of w to an element of A.

Given two worlds w_1 and w_2, an object of the dependent function type

$$w_1 \multimap w_2 \equiv (x : |w_2|) \to (w_1 \Rightarrow w_2@x)$$

is called a *world map* (p. 4). The intuition behind the notation $w_1 \multimap w_2$ is that an object of this type maps a realization of w_1 to a realization of w_2.

There is a category Wm of worlds and world maps, and this category has arbitrary products (Thms. 5, 6).

There is a full and faithful contravariant functor from the category Wm to the category Mnd of monads and monad morphisms (Thm. 7).

The notion of world generalizes the standard notion of *interface* (p. 6) familiar from object-oriented programming; and a world map

$$c : r \multimap p,$$

where r and p are interfaces (worlds), can be viewed as a *component* with required interface r and provided interface p.

Several natural applications of worlds and world maps to component-based development are given in Section III. For example, the problem of session state is given a very clean solution (p. 8).

Notation

This paper uses standard notation from functional programming and type theory, summarized in Table I. The usual conventions of functional programming are that application is left associative and binds higher than infix operations, and that arrows are right associative. The priority of infix operations should always be clear from the context, but, as a general rule, relational operators have lowest priority, then arrows, and finally arithmetic operators. For abstraction, I have revived the notation $\hat{x}\ b$ from Russell [18, p. 250]. As usual, the scope of the bound variable is as far right as possible.

In Martin-Löf's type theory, it is customary to employ the full Curry-Howard correspondence between propositions and sets, and treat the notions 'set' and 'prop' as synonymous. However, the gist of this paper can be appreciated without knowledge of the Curry-Howard correspondence.

doi:10.4304/jsw.7.5.1136-1148

TABLE I
SUMMARY OF TYPE-THEORETIC NOTATION FOR SETS, CANONICAL ELEMENTS, IMPORTANT OPERATIONS, AND THE LOGICAL CONNECTIVE CORRESPONDING TO THE SET UNDER THE CURRY-HOWARD CORRESPONDENCE.

Construct	Notation	Canonical elements	Operations	Logical connective
Dependent sum	$(\Sigma\ x : A)\ B\ x$	(a, b)	fst, snd	Existential quantifier
Dependent product	$(x : A) \to B\ x$	$\hat{x}\ b$	(*application*)	Universal quantifier
Disjoint union	$A + B$	left a, right b	(*pattern matching*)	Disjunction
Unit set	unit	$()$	—	True proposition
Empty set	$\emptyset$	—	abort	False proposition

Outline of paper

Section I contains a summary of the paper by Hancock and Setzer [10], with a new treatment of equality between interactive programs.

The main theoretical contribution of this paper, i.e., the notion of world map, is presented in Section II. This Section is somewhat technical and the details may be skipped at a casual reading of the paper.

The identification of worlds with interfaces and world maps with components is made in Section III. Moreover, several examples of how world maps can be used in programming are given in this Section.

In Section IV, worlds and world maps are related to work on containers, monads, and component-based development.

Finally, some conclusions are drawn in Section V.

The paper is rounded off by an Appendix with a note on program recursion and detailed proofs of the important theorems.

Throughout, the reader is assumed to be broadly familiar with Martin-Löf's type theory, but concepts such as monad, world, and interactive program will be explained.

I. THE MONAD OF A WORLD

Definition of monad

The importance of the category-theoretical concept of *monad* for semantics of programming languages was first explained by Moggi [17]: monads unify things like exceptions, global state, input and output.

In functional programming, a monad is typically defined as a type constructor M, with two operations, called return and bind, satisfying certain laws. Subsequently, return will be abbreviated 'ret', and the binary bind operation will be written $m \gg\!= f$. The following type-theoretic definition of the notion of *monad* will be used.[1]

A *monad* is a quadruple, with components

$$\begin{aligned} M &: \mathrm{set} \to \mathrm{set}, \\ (\doteq) &: (A : \mathrm{set}) \to M\,A \to M\,A \to \mathrm{prop}, \\ \mathrm{ret} &: (A : \mathrm{set}) \to A \to M\,A, \\ (\gg\!=) &: (A : \mathrm{set}) \to (B : \mathrm{set}) \to M\,A \to \\ &\qquad (A \to M\,B) \to M\,B, \end{aligned}$$

together with a number of proof objects (described below). The following abbreviations will be used:

$$\begin{aligned} a \doteq_A b &\equiv (\doteq)\ A\ a\ b, \\ \mathrm{ret}_A\ a &\equiv \mathrm{ret}\ A\ a, \\ p \gg\!=^B_A f &\equiv (\gg\!=)\ A\ B\ p\ f. \end{aligned}$$

Eventually, the subscript A and the superscript B will be dropped (though they can always be inferred from the context). A monad satisfies the three *monad laws*,

$$\begin{aligned} p \gg\!=^A_A \mathrm{ret}_A &\doteq_A p, \\ \mathrm{ret}_A\ a \gg\!=^B_A f &\doteq_B f\ a, \\ (p \gg\!=^B_A f) \gg\!=^C_B g &\doteq_C p \gg\!=^C_A (\hat{x}\ f\ x \gg\!=^C_B g), \end{aligned}$$

where $p : M\,A$, $a : A$, $f : A \to M\,B$, $g : B \to M\,C$, and the hat on the x denotes abstraction. These laws are called, respectively, *right identity*, *left identity*, and *associativity*. Moreover, the relation $(\doteq_A)$ is an equivalence relation, for any set A, and the bind operation is extensional, i.e.,

$$p \gg\!=^B_A f \ \doteq_B \ q \gg\!=^B_A g,$$

provided that

$$p \doteq_A q \text{ and } f\ x \doteq_B g\ x \text{ for all } x : A.$$

Thus, to fully define a monad, seven proof objects have to be provided, in addition to the quadruple: three to prove the monad laws, three to prove that equality is an equivalence relation, and one to prove that bind is extensional.

Definition of world

The notion of *world* is defined as follows by Hancock and Setzer [10, p. 320]. A *world* consists of a set C together with a family of sets R over the set C. An element c of the set C is a *command*, and the set $(R\ c)$ is the set of possible *responses* to the command c. That w is a world will be written $w : \mathrm{world}$.

If w is a world, the set of commands of w is denoted $|w|$ and the set of responses to a command c is written $w@c$. The world with commands C and responses R is written $(x : C, R\ x)$, i.e.,

$$\begin{cases} |(x : C, R\ x)| \equiv C & : \mathrm{set}, \\ (x : C, R\ x)@c \equiv R\ c & : \mathrm{set}. \end{cases}$$

A realization of a world w is an interactive device, external to type theory, which one can invoke with an element c of $|w|$, wait for a while, and get back an element of $w@c$. The concept of realization is not a part of the formal theory of worlds and word maps, and plays no rôle in it: the concept is only used to guide the reader's intuition. Informally, a world now becomes the "signature" of concrete interactive devices or the "type" of its realizations.

[1] The notion of monad used in functional programming is not the category-theoretical notion. Instead it is more akin to the notion of Kleisli triple, also from category theory. But there is a one-one correspondence between monads and Kleisli triples, so this ambiguity is mostly harmless.

Example

The simple world 'io' of console applications, supporting only input and output of character strings, is defined as follows. Its set of commands is called 'cio' and the response to a command c is the set $(\mathrm{rio}\ c)$. That is,

$$\mathrm{io} \equiv (x : \mathrm{cio}, \mathrm{rio}\ x) : \mathrm{world}.$$

The constant 'cio' has two constructors, $\mathrm{read} : \mathrm{cio}$, and $\mathrm{write} : \mathrm{string} \to \mathrm{cio}$. The constant 'rio' is defined by

$$\begin{cases} \mathrm{rio\ read} & \equiv & \mathrm{string} & : \mathrm{set}, \\ \mathrm{rio}\ (\mathrm{write}\ s) & \equiv & \mathrm{unit} & : \mathrm{set}. \end{cases}$$

That is, 'read' is a command that returns a string and $(\mathrm{write}\ s)$, where s is a string, is a command that returns an element of the one-element set 'unit' (i.e., it returns no information). Informally, the above information can be encoded by

$$\mathrm{io} \equiv \begin{cases} \mathrm{read} & :: \mathrm{string}, \\ \mathrm{write}\ (x : \mathrm{string}) & :: \mathrm{unit}. \end{cases}$$

A similar notation will be used whenever it is clear how to encode the "methods" of a world into a set of commands and a family of responses to commands.

Definition of interactive program

The notion of *interactive program* is also taken from Hancock and Setzer [10, p. 321]. Given that w is a world and that A is a set, one can form the set of *interactive programs* over this world with results in A. This set is written $w \Rightarrow A$ and has formation rule

$$\frac{w : \mathrm{world} \quad A : \mathrm{set}}{w \Rightarrow A : \mathrm{set}}\ .$$

The intuition behind the notation $w \Rightarrow A$ that an interactive program takes a realization of w and gives an element of A as result. How can this be translated into an inductive definition? First, the program needs not interact with the world, i.e., it may directly return element of A. This program will be written $(\mathrm{ret}\ a)$ and has introduction rule

$$\frac{w : \mathrm{world} \quad a : A}{\mathrm{ret}\ a : w \Rightarrow A}\ .$$

Next, the program must be able to interact with the world by invoking any command c and get access to the response x of the command. Put differently, given any command $c : |w|$, and any program $(t\,x)$, where x ranges over the set $w@c$, a new program $(\mathrm{invk}\ c\ t)$ can be formed. To execute this program, given a realization of the world, invoke the world on the command c and wait for the response r, then continue by executing $(t\ r)$. The above explanation justifies the inference rule

$$\frac{w : \mathrm{world} \quad c : |w| \quad t : w@c \to w \Rightarrow A}{\mathrm{invk}\ c\ t : w \Rightarrow A}\ ,$$

connecting the three notions *command*, *response*, and *program*.

Example

A program 'pw' over the 'io' world that asks for a password and compares it to a given password, returning a Boolean, would have type

$$\mathrm{pw} : \mathrm{string} \to \mathrm{io} \Rightarrow \mathrm{bool},$$

and can be defined by

$$\mathrm{pw}\ x \equiv \mathrm{invk}\ (\mathrm{write}\ \text{"enter password:"})\ (\hat{\cdot}\ \mathrm{invk\ read}\ (\hat{y}\ \mathrm{ret}\ (\mathrm{equals}\ x\ y))) : \mathrm{io} \Rightarrow \mathrm{bool}.$$

Clearly, this syntax is not optimal for writing large programs, but it can be simplified by nominal definitions, or, even better, by something similar to Haskell's do-notation.

The bind operation

The 'bind' operation, familiar from functional programming with monads, has typing rule

$$\frac{a : w \Rightarrow A \quad f : A \to w \Rightarrow B}{a \gg\!= f : w \Rightarrow B}$$

and is defined by the equations [10, p. 322]:

$$\begin{cases} (\mathrm{ret}\ a) \gg\!= f & \equiv & f\ a & : w \Rightarrow B, \\ (\mathrm{invk}\ c\ t) \gg\!= f & \equiv & \mathrm{invk}\ c\ (\hat{x}\ t\ x \gg\!= f) & : w \Rightarrow B. \end{cases}$$

Incidentally, the first of these equalities is the monad law of left identity, which thus holds up to definitional equality in the program monad.

The bind operation $p \gg\!= f$ has a natural interpretation in terms of program trees: it represents the tree p with all leaves $(\mathrm{ret}\ a)$ replaced by $(f\ a)$.

Equality between programs

The usual type-theoretic propositional equality is unsuitable for equality between programs, because it is too strict. Hancock and Setzer [10, p. 324] introduce non-well-founded programs before they investigate equality, and resort to bisimilarity for equality between programs. In contrast, the notion of equality proposed here is well-founded and a new primitive of type theory. The formation rule for program equality is given by

$$\frac{p : w \Rightarrow A \quad q : w \Rightarrow A}{p \doteq q : \mathrm{prop}}\ .$$

Neither bisimilarity, nor the equality relation $p \doteq q$, is substitutive.

There are two ways of demonstrating that two programs are equal, corresponding to the two forms of canonical programs. First, two programs $(\mathrm{ret}\ a)$ and $(\mathrm{ret}\ b)$ are equal if a and b are definitionally equal. Since definitional equality is implicit in all inference rules, the program equality between $(\mathrm{ret}\ a)$ and $(\mathrm{ret}\ b)$ is captured by the inference rule

$$\frac{a : A}{\mathrm{rret}\ a : \mathrm{ret}\ a \doteq \mathrm{ret}\ a}\ .$$

Next, let c and d be equal commands (elements of $|w|$), let s be a continuation for c and t is a continuation for d — the programs $(\mathrm{invk}\ c\ s)$ and $(\mathrm{invk}\ d\ t)$ ought to be equal if $(s\ x)$

and $(t\ x)$ are equal programs under the assumption that x is an element of the set $w@c \equiv w@d$. Again, since definitional equality is implicit in inference rules, this definition is captured by

$$\frac{c : |w| \quad s, t : w@c \to w \Rightarrow A \quad h : (x : w@c) \to s\ x \doteq t\ x}{\mathrm{rinvk}\ h : \mathrm{invk}\ c\ s \doteq \mathrm{invk}\ c\ t}\ .$$

This definition, as well as the definition of the program set, falls under the general scheme for inductive families laid down by Dybjer [6]. The elimination rule for program equality will not be spelled out. Instead, proofs involving program equality will use induction, often without explicit proof objects.

Proposition 1: Program equality is an equivalence relation.

Proof: The proofs of reflexivity and symmetry are straightforward. The proof of transitivity is trickier. We like to define the constant

$$\mathrm{trans}\ p\ q\ v\ r\ v' : p \doteq r,$$

where $p, q, r : w \Rightarrow A$, $v : p \doteq q$ and $v' : q \doteq r$. The proof is by double induction, first on v, and then on r. The details are deferred to the Appendix. □

Theorem 2 (Hancock and Setzer): Given that w is a world, $(w \Rightarrow)$ is a monad satisfying the monad laws (p. 2) up to program equality.

Proof: The notation $(w \Rightarrow)$ stands for the function of type $\mathrm{set} \to \mathrm{set}$ that takes a set A to the set $w \Rightarrow A$. As seen above, left identity holds up to definitional equality. Since program equality is reflexive, it holds up to program equality as well. Right identity and associativity are demonstrated by induction on p. The details are deferred to the Appendix. □

II. The Category of Worlds and World Maps

Definition of world map

Let w_1 and w_2 be worlds. The set of *world maps* from w_1 to w_2 will be written $w_1 \multimap w_2$. The intuition behind a world map $m : w_1 \multimap w_2$ is that it maps a realization of w_1 into a realization of w_2. That is, given that w_1 is realized, m must be able to provide a response to every command of w_2, possibly by interacting with the world w_1. This intuition motivates the definition

$$w_1 \multimap w_2 \equiv (x : |w_2|) \to w_1 \Rightarrow w_2@x : \mathrm{set}.$$

That is $(m\ c)$, where $c : |w_2|$, is an interactive program over the world w_1 giving as result an element of the set $w_2@c$.

Equality between world maps is defined by universal quantification over program equality. That is, if w_1 and w_2 are worlds, and $m, n : w_1 \multimap w_2$, the equality $m \mathrel{\dot{=}} n$ is defined by

$$m \mathrel{\dot{=}} n \equiv (x : |w_2|) \to m\ x \doteq n\ x : \mathrm{prop}.$$

Lifting

The most important property of a world map $m : w_1 \multimap w_2$ is that it can be used to interpret programs over w_2 in terms of programs over w_1. This process will be called *lifting* and the corresponding type-theoretic constant has typing rule

$$\frac{w_1, w_2 : \mathrm{world} \quad m : w_1 \multimap w_2 \quad A : \mathrm{set}}{\mathrm{lift}_A\ m : (w_2 \Rightarrow A) \to (w_1 \Rightarrow A)}\ .$$

This constant is defined by

$$\begin{cases} \mathrm{lift}_A\ m\ (\mathrm{ret}\ a) & \equiv & \mathrm{ret}\ a, \\ \mathrm{lift}_A\ m\ (\mathrm{invk}\ c\ t) & \equiv & m\ c \gg\!= (\hat{x}\ \mathrm{lift}_A\ m\ (t\ x)). \end{cases}$$

Lifting is extensional in both arguments, as 'bind' is extensional in both arguments.

Composition of world maps

World maps are composed using the inference rule

$$\frac{m : w_1 \multimap w_2 \quad n : w_2 \multimap w_3}{(m; n) : w_1 \multimap w_3}\ .$$

Note that the order of the arguments is the opposite of the usual order for composition of functions. Composition of world maps is defined by

$$(m; n)\ c \equiv \mathrm{lift}\ m\ (n\ c) : w_1 \Rightarrow w_3@c,$$

for $c : |w_3|$. Note that $(n\ \ c) : w_2 \Rightarrow w_3@c$. Since composition is defined in terms of lifting, it is extensional in both arguments.

Lemma 3: The identity map $\mathrm{id}_w : w \multimap w$, defined by

$$\mathrm{id}_w\ c \equiv \mathrm{invk}\ c\ \mathrm{ret} : w \Rightarrow w@c,$$

for $c : |w|$, is an identity with respect to composition.

Proof: Let $m : w_1 \multimap w_2$ and $n : w_2 \multimap w_1$. There are two things to show. First that

$$(\mathrm{id}_{w_1}; m)\ c \equiv \mathrm{lift}\ \mathrm{id}_{w_1}\ (m\ c)$$

is equal to $(m\ c)$, for $c : |w_2|$, and next that

$$\begin{aligned}(n; \mathrm{id}_{w_2})\ c &\equiv \mathrm{lift}\ n\ (\mathrm{id}_{w_2}\ c) \equiv \mathrm{lift}\ n\ (\mathrm{invk}\ c\ \mathrm{ret}) \\ &\equiv n\ c \gg\!= (\hat{x}\ \mathrm{lift}\ n\ (\mathrm{ret}\ x)) \equiv n\ c \gg\!= \mathrm{ret}\end{aligned}$$

is equal to $(n\ c)$, for $c : |w_1|$. The second equality follows directly from the monad law of right identity. To prove the first equality, we prove that $(\mathrm{lift}\ \mathrm{id}_w)$ is equal to the identity function. The proof is by induction on the (implicit) argument p. The case when $p \equiv \mathrm{ret}\ a$ is trivial. In case $p \equiv \mathrm{invk}\ c\ t$, the result follows from

$$\mathrm{lift}\ \mathrm{id}_w\ (\mathrm{invk}\ c\ t) \equiv \mathrm{invk}\ c\ (\hat{x}\ \mathrm{lift}\ \mathrm{id}_w\ (t\ x)),$$

the induction hypothesis, and 'rinvk'. □

Lemma 4: Composition of world maps is associative.

Proof: What we need to prove is that the two ways of composing up the world maps in

$$w_1 \ \text{—}f\text{—}\circ\ w_2 \ \text{—}g\text{—}\circ\ w_3 \ \text{—}h\text{—}\circ\ w_4$$

are equal. By definition of equality between world maps, it suffices to show that, for any command c of the world w_4, the two programs $((f; g); h)\ c$ and $(f; (g; h))\ c$ are equal. The left hand side is equal to $(\mathrm{lift}\ (f; g)\ (h\ \ c))$ and the right hand side is equal to $(\mathrm{lift}\ f\ (\mathrm{lift}\ g\ (h\ \ c)))$. Thus, it suffices to show that $(\mathrm{lift}\ (f; g)\ p)$ and $(\mathrm{lift}\ f\ (\mathrm{lift}\ g\ p))$ are equal programs for any $p : w_3 \Rightarrow A$. In case $p \equiv \mathrm{ret}\ a$, we have $\mathrm{lift}\ (f; g)\ (\mathrm{ret}\ a) \equiv \mathrm{ret}\ a : w_1 \Rightarrow A$, and

$$\mathrm{lift}\ f\ (\mathrm{lift}\ g\ (\mathrm{ret}\ a)) \equiv \mathrm{lift}\ f\ (\mathrm{ret}\ a) \equiv \mathrm{ret}\ a : w_1 \Rightarrow A,$$

for $a : A$. In case $p \equiv \text{invk}\ c\ d : w_3 \Rightarrow A$, where $c : |w_3|$ and $d : w_3@c \to w_3 \Rightarrow A$, we have

$$\text{lift}\ (f; g)\ (\text{invk}\ c\ d) \equiv \text{lift}\ f\ (g\ c) \gg\!= (\hat{x}\ \text{lift}\ (f; g)\ (d\ x)),$$

and

$$\text{lift}\ f\ (\text{lift}\ g\ (\text{invk}\ c\ d)) \equiv \text{lift}\ f\ (g\ c \gg\!= (\hat{x}\ \text{lift}\ g\ (d\ x))).$$

To show that these two programs are equal we use induction on $(g\ c)$ and associativity of bind. The details are deferred to the Appendix. □

Theorem 5: There exists a category Wm of worlds and world maps.[2]

Proof: By the previous two lemmata. □

Point-free programming

Recall that the intuition behind the notation $w_1 \multimap w_2$ is that an object of this type maps a realization of w_1 to a realization of w_2. Moreover, the intuition behind the notation $w_2 \Rightarrow A$ is that an object of this type maps a realization of w_2 to an element of A.

Realizations of worlds cannot be brought into the language of type theory, but one can program with realizations of worlds in a tacit or point-free way, using "combinators" like bind, composition, lifting, identity, and the projections and products of world maps defined below.

Families of worlds

Given that A is a set, a *family of worlds* Ω over A is a pair (C_Ω, R_Ω), where $C_\Omega : A \to \text{set}$ and $R_\Omega : (x : A) \to C_\Omega\ x \to \text{set}$. If $\Omega \equiv (C_\Omega, R_\Omega)$ is a family of worlds over A, and $a : A$, the world $(\Omega\ a)$ is, by definition, $(C_\Omega\ a, R_\Omega\ a)$. Let Ω be a family of worlds over a set A and define the world $(\Pi\ A\ \Omega)$ by

$$|\Pi\ A\ \Omega| \equiv (\Sigma\ x : A)\ |\Omega\ x| : \text{set},$$
$$(\Pi\ A\ \Omega)@(x, y) \equiv (\Omega\ x)@y : \text{set},$$

where $x : A$ and $y : |\Omega\ x|$. Note that this definition can be written

$$\Pi\ A\ \Omega \equiv$$
$$(z : (\Sigma\ x : A)\ |\Omega\ x|\ ,\ \ (\Omega\ (\text{fst}\ z))@(\text{snd}\ z)) : \text{world},$$

so there is no size problem, i.e., the family of response sets is not defined by pattern matching.

The projections $\pi_i : \Pi\ A\ \Omega \multimap \Omega\ i$, for $i : A$, are defined by

$$\pi_i\ b \equiv \text{invk}\ (i, b)\ \text{ret} : \Pi\ A\ \Omega \Rightarrow (\Omega\ i)@b,$$

for $b : |\Omega\ i|$. This definition is well-formed since the set $(\Pi\ A\ \Omega)@(i, b)$ is equal to the set $(\Omega\ i)@b$.

Theorem 6: The product of a family of worlds is a categorical product in the category of worlds and world maps.

[2]To be precise, this Theorem establishes that Wm is an *E-category* — the type-theoretical notion of category — with the additional *caveat* that its class of objects is not a set or type, but the telescope 'world'. The reader is referred to a paper by Buisse and Dybjer [5] for the exact definition of the notion of E-category.

Proof: Let $A : \text{set}$ and a family of worlds Ω over A be fixed. Assume that w is an arbitrary world and that $g_i : w \multimap \Omega\ i$ for $i : A$. We must show that there is a unique world map $\langle g_j \rangle_j^A : w \multimap \Pi\ A\ \Omega$ such that $(\langle g_j \rangle_j^A ; \pi_i) \doteq g_i$, for any $i : A$. Recall that

$$w \multimap \Pi\ A\ \Omega \equiv$$
$$(x : (\Sigma\ y : A)\ |\Omega\ y|) \to w \Rightarrow (\Pi\ A\ \Omega)@x : \text{set}.$$

Define $\langle g_j \rangle_j^A$ by

$$\langle g_j \rangle_j^A\ (i, b) \equiv g_i\ b : w \Rightarrow (\Omega\ i)@b,$$

for $i : A$ and $b : |\Omega\ i|$. For an arbitrary $h : w \multimap \Pi\ A\ \Omega$, $i : A$, and $c : |\Omega\ i|$, we have $(h; \pi_i) : w \multimap \Omega\ i$, and

$$(h; \pi_i)\ c \equiv \text{lift}\ h\ (\pi_i\ c) \equiv \text{lift}\ h\ (\text{invk}\ (i, c)\ \text{ret}) \equiv$$
$$h\ (i, c) \gg\!= (\hat{x}\ \text{lift}\ h\ (\text{ret}\ x)) \equiv h\ (i, c) \gg\!= \text{ret}$$
$$: w \Rightarrow (\Omega\ i)@c,$$

whence, by right identity, the proposition

$$(h; \pi_i)\ c \doteq h\ (i, c) : \text{prop}$$

is true. This shows both that $\langle g_j \rangle_j^A$ is unique up to equality between world maps, and that $(\langle g_j \rangle_j^A ; \pi_i) \doteq g_i$. □

Binary products

It may seem like a paradox, but Wm does not have all binary products. This is because arbitrary products are defined in terms of families of worlds, and given two worlds w_1 and w_2, there is no guarantee that there exists a family of worlds Ω over $\{1, 2\}$ with $\Omega\ 1 \equiv w_1$ and $\Omega\ 2 \equiv w_2$.

However, when both w_1 and w_2 are *small*, the binary product is guaranteed to exist. A world w is small if there is a code for $|w|$ and a family of codes for $w@x$ in a suitable type-theoretic universe. When w_1 and w_2 are small worlds, the usual notation $w_1 \times w_2$ is used for their product. If the objects of the category Wm are restricted to small worlds, the resulting category has binary products.

Comparison with the category of monads

The category Wm will now be compared to the category $\mathrm{Mnd}^{\mathrm{op}}$ of monads and reverse monad morphisms. A monad morphism between monads M and N is a family of functions $f_A : M\ A \to N\ A$ that respect the return and bind operations up to the monad equalities. Two morphisms f and g in Mnd are equal if $f_A\ x \doteq g_A\ x$ for all sets A, and $x : M\ A$ — this will be written $f \cong g$. It will be accepted without proof that there exists an E-category of monads and monad morphisms.

Theorem 7: There exists a full and faithful functor from the category Wm to the category $\mathrm{Mnd}^{\mathrm{op}}$ of monads and reverse monad morphisms, mapping a world w to the monad $(w \Rightarrow)$ and a world map $m : w_1 \multimap w_2$ to the monad morphism $(\text{lift}\ m)$.

Proof: There are several things to establish. First, that the above mapping defines a functor from Wm to $\mathrm{Mnd}^{\mathrm{op}}$. The proof that id_w is an identity with respect to composition of world maps also shows that $\text{lift}\ \mathrm{id}_w$ is equal to the

identity function. The proof that composition of world maps is associative shows that $\text{lift}\ (f;g) \cong (\text{lift}\ f) \circ (\text{lift} g)$, where $(\cong)$ denotes equality of monad morphisms. To establish that we have a functor, one more thing is required, viz., that $(\text{lift}\ m)$ indeed is a monad morphism, i.e., that the proposition

$$\text{lift}_B\ m\ (n \gg\!= f) \doteq \text{lift}_A\ m\ n \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) : \text{prop}$$

is true, for any $n : w_2 \Rightarrow A$ and $f : A \to w_2 \Rightarrow B$. The proof is by induction on n. The details are deferred to the Appendix.

To show the functor is *full*, i.e., that any morphism between world-based monads arises from a world map by lifting, assume that w_1 and w_2 are worlds, and that we have monad morphism $f_A : (w_2 \Rightarrow A) \to (w_1 \Rightarrow A)$, with $f_A\ (\text{ret}\ a) \doteq (\text{ret}\ a)$, and $f_B\ (a \gg\!= g) \doteq f_A\ a \gg\!= (\hat{x}\ f_B\ (g\ x))$. If $c : |w_2|$ and $t : w_2@c \to w_1 \Rightarrow B$, we have

$$f_B\ (\text{invk}\ c\ t) \equiv f_B\ (\text{invk}\ c\ \text{ret} \gg\!= t) \doteq f_{w_2@c}\ (\text{invk}\ c\ \text{ret}) \gg\!= (\hat{x}\ f_B\ (t\ x)).$$

If we make the definition $m\ c \equiv f_{w_2@c}\ (\text{invk}\ c\ \text{ret})$, the definition of 'lift' gives extensional equality between f_B and $\text{lift}_B\ m$, as $\text{lift}_A\ m\ (\text{ret}\ a) \equiv \text{ret}\ a$, and

$$\text{lift}_B\ m\ (\text{invk}\ c\ t) \equiv m\ c \gg\!= (\hat{x}\ \text{lift}_B\ m\ (t\ x)).$$

That is, f satisfies the equations that define $\text{lift}\ m$, but up to program equality instead of definitional equality.

To show that the functor is *faithful*, assume that $\text{lift}\ m \cong \text{lift}\ n$, for world maps $m, n : w_1 \multimap w_2$. For any $c : |w_2|$, we have $\text{lift}\ m\ (\text{invk}\ c\ \text{ret}) \equiv m\ c \gg\!= \text{ret}$, and similarly for n. It follows by identity that $m\ c \doteq n\ c$, as required. □

III. Programming with world maps

Component-based software engineering

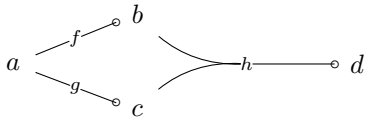
One of the main problems in software architecture is how to gain a composable understanding of large software systems.

> [7]: As the size of software systems increases, the algorithms and data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organization of the overall system — the software architecture — presents a new set of design problems.

The composability problem is often approached through the notion of component. To see how worlds and world maps can be applied to the field of software architecture, make the identifications

$$\begin{aligned} \text{world} &\equiv \text{interface}, \\ \text{world map} &\equiv \text{component}. \end{aligned}$$

Just by making these identifications, interfaces and components form a category. This ensures that well-behaved component diagrams will have a unique interpretation. For example, associativity ensures that there is a unique way of composing three morphisms f, g, h, where $f : a \multimap b$, $g : b \multimap c$, and $h : c \multimap d$. A component diagram like



is interpreted as follows: a, b, c, and d are interfaces; $f : a \multimap b$, $g : a \multimap c$, and $h : b \times c \multimap d$ are components; and the diagram stands for the component $(\langle f, g \rangle; h) : a \multimap d$.

If $c : r \multimap p$ is a component, r is called its *required interface* and p is called its *provided interface*. That is, p is the interface that clients of c use, and r is the interface that c requires to be able to provide its functionality. From an object-oriented point of view, the component c is a stateless *adapter* that implements the interface p in terms of the interface r.

One distinguishing feature of this notion of component is that components (world maps) are *stateless*, i.e., they have no *internal state*. Nevertheless, a stateful component $c : r \multimap_S p$, with states drawn from a set S, can be represented by $c : r \times \text{st}_S \multimap p$ (the world st_S is defined below). For a large composite of stateful components, this means that the internal state of every component will be visible in the required interface of the composite component. This has clear advantages for monitoring, debugging, and performance analysis, as it is easy to inspect a running component's state.

The disadvantage is similar to the disadvantage that pure functional programming had over imperative programming before the introduction of monads, viz., that the state had to be mentioned everywhere and manipulated explicitly.

World-based equivalents of standard monads

Many basic worlds can be derived from Haskell's monad system.

Standard monads, like the state and list monads, cannot be directly represented by world-based monads. However, the following trick, due to Swierstra [19, p. 431], can be used. Even though the state monad is not *isomorphic* to a world-based monad, it is *covered* by one. That is, there is a monad epimorphism from the monad of the world

$$\text{st}_S \equiv \begin{cases} \text{get} & :: S, \\ \text{set}\ (x : S) & :: \text{unit} \end{cases}$$

to the standard state monad. This epimorphism, $\text{state} : (\text{st}_S \Rightarrow A) \to (S \to A \times S)$, is defined by

$$\begin{cases} \text{state}\ (\text{ret}\ a)\ s & \equiv & (a, s) & : A \times S, \\ \text{state}\ (\text{invk}\ (\text{set}\ t)\ d)\ s & \equiv & \text{state}\ (d\ ())\ t & : A \times S, \\ \text{state}\ (\text{invk}\ \text{get}\ d)\ s & \equiv & \text{state}\ (d\ s)\ s & : A \times S. \end{cases}$$

This can be generalized to a function $\text{state}_w : (\text{st}_S \times w \Rightarrow A) \to S \to w \Rightarrow A \times S$. Similarly, the list monad is covered by the monad of the world

$$\text{lst} \equiv \begin{cases} \text{join} & :: \text{unit} + \text{unit}, \\ \text{empty} & :: \emptyset, \end{cases}$$

TABLE II
WORLDS CORRESPONDING TO SOME OF THE STANDARD MONADS OF HASKELL, WITH THEIR SETS OF COMMANDS AND RESPONSES.

Monad / World	w	$\|w\|$	$w@x$
Exception	exn_E	E	$\emptyset$
Reader	rd_U	unit	U
Writer	wr_U	U	unit
Terminal (null)	term	$\emptyset$	$\emptyset$

by means of the monad epimorphism $\mathrm{flatten} : (\mathrm{lst} \Rightarrow A) \to A\ \mathrm{list}$, defined by

$$\begin{cases} \mathrm{flatten}\ (\mathrm{ret}\ a) & \equiv & [a], \\ \mathrm{flatten}\ (\mathrm{invk\ join}\ d) & \equiv & d\ (\mathrm{left}\ ()) \oplus d\ (\mathrm{right}\ ()), \\ \mathrm{flatten}\ (\mathrm{invk\ empty}\ d) & \equiv & [], \end{cases}$$

where $\oplus$ denotes list concatenation.

The exception, reader, and writer monad are isomorphic to world-based monads. The corresponding worlds, exn_E, rd_U, and wr_U, have command and response sets given by Table II. The last line of this Table introduces the *terminal* world, which is a terminal object in the category Wm.

The response set of these worlds does not depend on the command. However, when combining two or more such worlds using the Π-construction, the response set of the resulting world depends on the command. This fits with the identification of worlds and interfaces, made above: an interface is the (world) product of its methods (also represented by worlds).

Event-driven programs

The approach to interactive programming used in Haskell, and extended to Martin-Löf's type theory by Hancock and Setzer [10], is based on the idea that a deployable program is an object of type $w \Rightarrow \mathrm{unit}$, for a suitable world w (e.g., $w \equiv \mathrm{io}$, def. on p. 3) — much like the 'main' procedure of a C program. Hancock and Setzer are forced to introduce non-well-founded 'while' programs to be able to write programs that run indefinitely.

I envision that platforms will execute components in customizable *component containers*. An interactive program of type $\mathrm{io} \Rightarrow \mathrm{unit}$ is replaced by an *event-driven* program of type $\mathrm{wr} \multimap \mathrm{wr}$ (where $\mathrm{wr} \equiv \mathrm{wr}_{\mathrm{string}}$), i.e., with a world map or component.

For example, a very simple platform, supporting only command line applications, could have a component container that executes components of type $\mathrm{st}_S \times \mathrm{wr} \multimap \mathrm{wr}$, for any (small) set S of states. This is sufficient to implement, e.g., a calculator. A simple event-driven program 'main' of type $\mathrm{st}_{\mathrm{nat}} \times \mathrm{wr} \multimap \mathrm{wr}$ is defined by

$$\begin{aligned} \mathrm{main}\ x \equiv\ & \mathrm{invk}\ (\mathrm{left}\ \mathrm{get}) \gg\!= \hat{z}\ \mathrm{invk}\ (\mathrm{left}\ (\mathrm{set}\ (z+1))) \\ & \gg\!= \hat{\cdot}\ \mathrm{invk}\ (\mathrm{right}\ (\mathrm{greeting}\ x\ z)) \\ & : \mathrm{st}_{\mathrm{nat}} \times \mathrm{wr} \Rightarrow \mathrm{unit}, \end{aligned}$$

for $x : \mathrm{string}$, where $\mathrm{greeting} : \mathrm{string} \to \mathrm{nat} \to \mathrm{string}$ is defined by

$$\begin{aligned} &\mathrm{greeting}\ x\ z \equiv \\ &\quad \text{"Hello "} \oplus x \oplus \text{", you have number "} \oplus \mathrm{str}\ z : \mathrm{string}. \end{aligned}$$

I leave to the reader to figure out what this program actually does. It is easy to construct more elaborate examples of components.

If a component needs to wake up at regular intervals (e.g., when polling), it can add the interface $\mathrm{wr}_{\mathrm{unit}}$ to its provided interfaces and let the component container provide it with "ticks" at specified intervals.

Short circuiting components

Given a component $c : s \times r \multimap s \times p$, one may like to "short circuit" the component c by, as it were, connecting the provided interface s to the required interface s. This is of course not possible in general, but it is not too difficult to come up with a suitable proof obligation $(\mathrm{scable}\ c)$. Given any command $d : |p|$, the program $(c; \pi_2)\ d : s \times r \Rightarrow p@d$ must be *accessible* with respect to a relation that orders programs according to how many times they produce commands from s when piped through the component $(c; \pi_1) : s \times r \multimap s$. Given a proof object $a : \mathrm{scable}\ c$, the short circuited component $(\mathrm{sc}\ c\ a) : r \multimap p$ can be defined by recursion on the proof of the accessibility predicate.

Asynchronous invocation

Consider a world (interface) a on which commands must be invoked asynchronously. Let h be the world that must handle the responses to the commands. Think, e.g., of a as a world for making HTTP requests and of h as a world for updating a graphical user interface.

Define a new world $(\mathrm{asynch}\ a\ h)$ that represents asynchronous invocations of commands from a with responses handled by h. A command of the world $(\mathrm{asynch}\ a\ h)$ consists of a command $c : |a|$ together with a response handler of type $a@c \to h \Rightarrow \mathrm{unit}$, that takes the response to c and gives a program over the world h. The response set of any command of $(\mathrm{asynch}\ a\ h)$ is the unit set. That is,

$$\begin{aligned} &\mathrm{asynch}\ a\ h \equiv \\ &\qquad (x : (\Sigma\ c : |a|)\ a@c \to h \Rightarrow \mathrm{unit}\ ,\ \mathrm{unit}) : \mathrm{world}. \end{aligned}$$

A program over the world $(\mathrm{asynch}\ a\ h)$ can be translated into a program over the world $a \times h$, i.e., an asynchronous program can be translated into a synchronous one using a world map (component) of type

$$\begin{aligned} &(a \times h) \multimap (\mathrm{asynch}\ a\ h) \equiv \\ &\quad ((\Sigma\ c : |a|)\ a@c \to h \Rightarrow \mathrm{unit}) \to (a \times h) \Rightarrow \mathrm{unit} : \mathrm{set}. \end{aligned}$$

To find its definition, assume that $c : |a|$ and $f : a@c \to h \Rightarrow \mathrm{unit}$, and observe that

$$\mathrm{invk}\ (\mathrm{left}\ c)\ (\hat{x}\ \mathrm{lift}\ \pi_2\ (f\ x)) : (a \times h) \Rightarrow \mathrm{unit},$$

where $\pi_2 : (a \times h) \multimap h$. This gives the required definition of the translation.

A component container can run components with the interface $(\mathrm{asynch}\ a\ h)$ amongst its required interfaces and execute commands over this interface asynchronously.

Session state

This example will demonstrate some of the "plasticity" of programming with dependent types. A conventional programming language (e.g., Java or C#) would have to use reflection to define the 'run' function.

A publicly available interface for a message board, with challenge-response authentication, could expose an interface along the lines of

$$\text{service} \equiv \begin{cases} \text{challenge} & :: \text{challenge}, \\ \text{response } (\mathit{res} : \text{response}) & :: \text{cookie}, \\ \text{list } (c : \text{cookie}) & :: \text{msgid list}, \\ \text{get } (c : \text{cookie}, \mathit{id} : \text{msgid}) & :: \text{string option}, \\ \text{put } (c : \text{cookie}, \mathit{msg} : \text{string}) & :: \text{msgid}, \\ \text{rm } (c : \text{cookie}, \mathit{id} : \text{msgid}) & :: \text{bool}. \end{cases}$$

where 'challenge', 'response', 'cookie', and 'msgid' are suitable datatypes (here left anonymous), and 'list' and 'option' are type-constructors like in ML.

It is somewhat cumbersome to program over an interface like this, and, ideally, one would like to use an interface like

$$\text{session} \equiv \begin{cases} \text{list} & :: \text{msgid list}, \\ \text{get } (\mathit{id} : \text{msgid}) & :: \text{string option}, \\ \text{put } (\mathit{msg} : \text{string}) & :: \text{msgid}, \\ \text{rm } (\mathit{id} : \text{msgid}) & :: \text{bool}. \end{cases}$$

instead, as the cookie will remain the same for an entire session.

Assuming that a cookie is given, a program over the interface 'session' can be transformed into a program over the interface 'service', by means of the function

$$\text{ss} : \text{cookie} \to (\text{session} \Rightarrow a) \to (\text{service} \Rightarrow a),$$

defined by

$$\begin{aligned} \text{ss } c\ (\text{ret } a) &\equiv \text{ret } a, \\ \text{ss } c\ (\text{invk list } t) &\equiv \text{invk } (\text{list } c)\ (\hat{x}\,\text{ss } c\ (t\ x)), \\ \text{ss } c\ (\text{invk } (\text{get } \mathit{id})\ t) &\equiv \text{invk } (\text{get } c\ \mathit{id})\ (\hat{x}\,\text{ss } c\ (t\ x)), \\ \text{ss } c\ (\text{invk } (\text{put } \mathit{msg})\ t) &\equiv \text{invk } (\text{put } c\ \mathit{msg})\ (\hat{x}\,\text{ss } c\ (t\ x)), \\ \text{ss } c\ (\text{invk } (\text{rm } \mathit{id})\ t) &\equiv \text{invk } (\text{rm } c\ \mathit{id})\ (\hat{x}\,\text{ss } c\ (t\ x)). \end{aligned}$$

That is, $(\text{ss } c)$ simply "fills in" the cookie c as the first argument of all methods.[3]

To complete the example, assume that the challenge-response protocol requires the client to respond to a challenge with a hash of its password concatenated with the challenge. Now given a password p and a session program s of type $\text{session} \Rightarrow a$, for some type a, we ought to be able to construct a program $(\text{run } p\ s)$ of type $\text{service} \Rightarrow a\ \text{option}$ that tries to log in, and, if successful, executes the program s, and, if unsuccessful, returns 'none'. The function 'run' should have type

$$\text{run} : \text{string} \to (\text{session} \Rightarrow a) \to (\text{service} \Rightarrow a\ \text{option}).$$

[3]The attentive reader may note that function 'ss' is easy to generalize to the case when we have a "container morphism" (p. 9) between two containers (worlds).

Its definition is given by

$$\begin{aligned} \text{run } p\ s \equiv\ & \text{invk challenge} \\ & (\hat{e}\ \text{invk } (\text{response } (\text{hash } (e \oplus p)))\ (\text{case } s)), \end{aligned}$$

where

$$\begin{aligned} \text{case } s\ \text{none} &\equiv \text{ret none} \\ \text{case } s\ (\text{some } c) &\equiv \text{ss } c\ s \gg\!= (\hat{x}\ \text{ret } (\text{some } x)). \end{aligned}$$

Unit testing

Object-oriented programs are often difficult to test. Ideally, one would like small unit tests, tightly coupled with the class being tested. This is typically accomplished by a combination of *mock objects* and *dependency injection*.

Component-based programs that are written according to the paradigm proposed in this paper are already prepared for this kind of testing. For example, given a component $c : r \multimap p$, the *mocking* would consist of implementing r in terms of some simple interface, like the terminal interface or an interface providing random numbers. The unit tests have type $p \Rightarrow \text{bool}$. If we combine the mock $m : \text{test} \multimap r$ with the component c and a unit test t, we get

$$\text{lift } (m; c)\ t : \text{test} \Rightarrow \text{bool}.$$

This object is directly testable. The interface 'test' can either be the terminal (empty) interface, or an interface providing methods for generating random data.

IV. Related work

Containers

The notion of *world* is related to the notion of *container*, introduced by Hoogendijk and de Moor [11]. The connection between containers and type theory is explained by Abbott, Altenkirch, and Ghani [1]. A container in the latter's sense is nothing but a world, i.e,

$$\text{container} \equiv \text{world}.$$

Given a world w, the *container functor* (endofunctor on the category of sets) corresponding to w is denoted $[\![w]\!]$ and is defined by

$$[\![w]\!]\ X \equiv (\Sigma\ x : |w|)\ w@x \to X : \text{set}.$$

The use of the word container is explained as follows. Let a container $w \equiv (x : S, P\,x)$ be given. The set S is called the set of *shapes* and $(P\,s)$ is called the set of *positions* in shape $s : S$. An element of $[\![w]\!]\ X$ consists of a shape $s : |w|$ and a function $f : w@s \to X$ that assigns an element of X to each position in shape s.

Monads

As mentioned in the introduction, monads are difficult to combine. One positive result is that the category Mnd, of monads and monad morphisms, has binary coproducts [14], but the construction is nontrivial to implement as it involves a quotient. Ghani and Uustalu [8] give a simpler construction

for the coproducts of two *ideal monads*. Moreover, it is easy to take the coproduct of *free monads*, and this construction is at the heart of the approach to programming with monads suggested by Swierstra [19].

In general, a *free* structure M over some underlying structure S is the "simplest" structure such that S can be "embedded" in M. Examples include the free group and free monoid over a set S of generators. A monad M is called *free*, over an underlying endofunctor F, if M is the "simplest" monad that admits a natural transformation from F to M. If the free monad exists, it is the least fixed point of the recursive equation

$$M\,A \cong A + F\,(M\,A),$$

cf., Ghani and Uustalu [8, Prop. 2.7].

It is not difficult to see that the monad $(w \Rightarrow)$ is the least fixed point for the recursive equation corresponding to the endofunctor $[\![w]\!]$, i.e.,

$$w \Rightarrow A \;\cong\; A + (\Sigma\, x : |w|)\; w@x \to w \Rightarrow A,$$

where 'left' corresponds to 'ret' and 'right' corresponds to 'invk'. Thus, every world-based monad is free.

Every world-based monad is free, and every free monad is ideal, so the category of world-based monads and monad morphisms is a subcategory of the category of (free/ideal) monads and monad morphisms. From this perspective, world-based monads are less general than free monads. The constructions of Swierstra [19] are formulated in terms of free monads, but most of them work equally well over world-based monads.

Strong functional programming

Martin-Löf's type theory is not only *pure*, but also *total*, i.e., all functions are terminating. This gives rise to what Turner [20] calls *strong* functional programming. In a total functional programming language, IO is typically approached through coinductive sets, to allow for programs that run indefinitely. For example, Hancock and Setzer [10] introduce non-well-founded IO-trees into Martin-Löf's type theory. However, the approach suggested in this paper is entirely based on well-founded sets.

Container morphisms

As pointed out to me by Peter Hancock, not only is the notion of world related to the notion of container, but the notion of world map is also related to the notion of container morphism.

Given two containers (worlds) c and d, the set of container morphisms $(\mathrm{Cont}\; c\; d)$ from c to d is given by

$$\mathrm{Cont}\; c\; d \equiv (\Sigma\, f : |c| \to |d|)\,(x : |c|) \to d@(f\; x) \to c@x.$$

Moreover, the free monad of a container c can be represented by the container c^*, i.e., the monad $(c \Rightarrow)$ is isomorphic to the monad $[\![c^*]\!]$. The "star" of a container is defined by $c^* \equiv (x : c \Rightarrow \mathrm{unit}\,,\; \mathrm{path}_c\; x) : \mathrm{world}$, where $\mathrm{path}_c : (c \Rightarrow \mathrm{unit}) \to \mathrm{set}$ is a function that gives the set of paths through a given tree that end with a "return" node. *Caveat*: the path function can only be defined for small worlds, unless type theory is strengthened with a new construct.

Using these two notions, there is an isomorphism $w_1 \multimap w_2 \;\cong\; \mathrm{Cont}\; w_2\; w_1^*$, for a suitable (extensional) notion of isomorphism. By the axiom of choice, used backwards,

$$\begin{aligned}\mathrm{Cont}\; c\; d &\equiv (\Sigma\, f : |c| \to |d|)\,(x : |c|) \to d@(f\; x) \to c@x \\ &\cong (x : |c|) \to (\Sigma\, y : |d|)\; d@y \to c@x \\ &\equiv (x : |c|) \to [\![d]\!]\; (c@x).\end{aligned}$$

Now, since $(w_1 \Rightarrow w_2@x)$ is isomorphic to $[\![w_1^*]\!]\; (w_2@x)$, the set $w_1 \multimap w_2$ is isomorphic to the set $\mathrm{Cont}\; w_2\; w_1^*$.

Component-based software engineering

The first use of category theory for the purpose of large scale system construction seems to be Goguen [9]. However, in that work, components are objects of the category and morphisms of the category represent communication between components. The idea of considering a component as a morphism between interfaces can be found in a PhD thesis due to Barbosa [4, Ch. 5].

Event-driven programming

According to Lauer and Needham [13], there is a certain duality between (operating) systems based on message passing ("events") and systems based on threads.

The programs presented in this paper are event-driven in the naïve sense that the flow of control is determined by user actions translated into invocations of methods on the provided interface. However, they are *not* based on message passing in the sense of Lauer and Needham.

In fact, concurrent execution of component-based programs fits best with the threaded model of execution. Nothing prevents an interactive program of type $a \Rightarrow i$ from blocking on certain operations. For most applications, threads provides the cleanest abstraction.

However, a component container can implement the required interface of a component using message passing without affecting the intended semantics of the component-based programming paradigm. Cf., the example of asynchronous invocation given above.

The set of wellorderings

The program set is a generalization of the W-set introduced by Martin-Löf [16, p. 79]. The following nominal definitions can be used to define the W-set and its associated constructor and destructor in terms of the program set:

$$\begin{cases} \mathrm{W}(C,R) & \equiv & (C,R) \Rightarrow \emptyset, \\ \sup(c,t) & \equiv & \mathrm{invk}\; c\; t, \\ \mathrm{T}(p,d) & \equiv & \mathrm{pgrec}\; \mathrm{abort}\; d\; p, \end{cases}$$

where 'pgrec' is defined in the Appendix and $\mathrm{abort} : \emptyset \to A$ is the polymorphic destructor of the empty set. The usual computation rule for $\mathrm{T}(p,d)$ can be derived using the computation rules for 'pgrec', given below. However, the program set cannot be defined in terms of the W-set, as 'pgrec' cannot

be defined (its definition would require that every function $\emptyset \to X$ is definitionally equal to 'abort').

An element of the set $\mathrm{W}(C, R)$ is an intuitionistic wellordering. A wellordering is a well-founded tree with the contents of the nodes drawn from the set C and branching factor $(R\ x)$ for a node with contents x. This interpretation carries over to the program set $w \Rightarrow A$, with the additional interpretation of A as the set from which the contents of leaf nodes is drawn.

It is interesting to review what Martin-Löf [15, p. 172] wrote: "The transfinite recursion form $\mathrm{T}(p, d)$ has not yet found any applications in programming. It has, as far as I know, no counterpart in other programming languages." Interactive programming is an application of transfinite recursion!

V. Conclusion and future work

This paper presents a new paradigm, based on world maps, to component-based development in Martin-Löf's type theory. The constructions are made in a fully intensional framework, and thus directly implementable — something which is not always the case with constructions in category theory.

Although based on inductive sets, world maps can represent (event-driven) programs that run indefinitely. Using the monad based approach to IO, the 'main' program has type $\mathrm{main} : \mathrm{io} \Rightarrow \mathrm{unit}$, i.e., it is a program over the 'io' monad without significant return value. Under the new paradigm, the 'main' program would instead have type $\mathrm{main} : \mathrm{out} \multimap \mathrm{in}$, where 'in' is the world of events that the program responds to and 'out' is the world the program uses to perform its job.

Thus, it is possible to view 'main' as a *component* with *provided interface* 'in' and *required interface* 'out'. The operating system *provides* the component's *required* functionality and propagates events to the component's provided interface. In this way, Martin-Löf's type theory is sufficient for systems programming, even without the addition of coinductive sets. Event-driven programming fits hand in glove with Martin-Löf's type theory, because event handlers are supposed to be terminating.

Several applications of the proposed component-based paradigm were given, but larger scale experiments are left as future work. Eventually, this component-based paradigm should be quantitatively compared to the object-oriented paradigm with respect to measures such as testability, reusability, and composability.

Acknowledgments

This work was supported by EPSRC grant EP/G03012X/1. Thanks to Peter Hancock for valuable comments on a draft of this paper. Thanks also to the anonymous reviewers who suggested significant improvements.

Appendix

Program recursion

Functions on wellorderings are defined by transfinite recursion. The corresponding recursion principle for interactive programs will be called *program recursion*. The constant 'pgrec', for program recursion, has typing rule

$$\frac{\begin{array}{l} D : (w \Rightarrow A) \to \mathrm{set} \\ r\ \ : (x : A) \to D\ (\mathrm{ret}\ x) \\ i\ \ : (x : |w|) \to (y : w@x \to w \Rightarrow A) \to \\ \qquad\qquad ((u : w@x) \to D\ (y\ u)) \to D\ (\mathrm{invk}\ x\ y) \\ p\ \ : w \Rightarrow A \end{array}}{\mathrm{pgrec}_D\ r\ i\ p : D\ p}\ .$$

The introduction of the constant 'pgrec' is justified by the computation rules

$$\begin{cases} \mathrm{pgrec}_D\ r\ i\ (\mathrm{ret}\ a) & \equiv & (r\ a), \\ \mathrm{pgrec}_D\ r\ i\ (\mathrm{invk}\ c\ t) & \equiv & i\ c\ t\ (\hat{x}\ \mathrm{pgrec}_D\ r\ i\ (t\ x)), \end{cases}$$

where the two sides of the first equation have type $D\ (\mathrm{ret}\ a)$, and the two sides of the second equation both have type $D\ (\mathrm{invk}\ c\ t)$. Note that the recursive invocation of 'pgrec' is on the program $(t\ x)$ which is smaller than $(\mathrm{invk}\ c\ t)$, as the program tree is well-founded.

In this paper, definitions by program recursion are given in equational form — but any such definition can be translated into an equivalent definition in terms of 'pgrec'. For example, using the constant 'pgrec', the definition of $\gg\!=$ (p. 3) becomes

$$a \gg\!= f \equiv \mathrm{pgrec}\ f\ (\hat{c}\ \hat{}\ \hat{n}\ \mathrm{invk}\ c\ n)\ a : w \Rightarrow B,$$

which, admittedly, is less readable than the original definition.

Proof of Proposition 1

The proofs of reflexivity and symmetry are straightforward. The proof of transitivity is trickier. Let a world w be fixed, and let P stand for the set $w \Rightarrow A$. We like to define the constant

$$\mathrm{trans}\ p\ q\ v\ r\ v' : p \doteq r,$$

where $p, q, r : P$, $v : p \doteq q$ and $v' : q \doteq r$. The first lemma is that program of invoke-form cannot be equal to a program on ret-form,

$$\mathrm{lemma}_1\ c\ s\ a : (\mathrm{invk}\ c\ s \doteq \mathrm{ret}\ a) \to \emptyset,$$

where $c : |w|$, $s : w@c \to P$, and $a : A$. The proof of this lemma is straightforward. The next lemma is that, if two programs of invoke-form are equal, their commands are propositionally equal,

$$\mathrm{lemma}_2\ c\ s\ d\ t : (\mathrm{invk}\ c\ s \doteq \mathrm{invk}\ d\ t) \to (c = d),$$

where $c, d : |w|$, $s : w@c \to P$, and $t : w@d \to P$. Moreover, if two programs of invoke-form are equal, their continuations are extensionally equal,

$$\begin{array}{l} \mathrm{lemma}_3\ c\ s\ d\ t : (p : \mathrm{invk}\ c\ s \doteq \mathrm{invk}\ d\ t) \to \\ \qquad (r : w@c) \to s\ r \doteq t\ (\mathrm{subst}_R\ (\mathrm{lemma}_2\ c\ s\ d\ t\ p)\ r), \end{array}$$

where c, d, s, and t are as above. Next, two programs where the commands are equal and the continuations are equal for all responses are equal,

$$\mathrm{lemma}_4\ c\ s\ d\ t\ l\ m : \mathrm{invk}\ c\ s \doteq \mathrm{invk}\ d\ t,$$

where c, d, s, and t are as above, $l : c = d$, and $m : (x : w@c) \to s\ x \doteq t\ (\mathrm{subst}_R\ l\ x)$. None of these lemmata are difficult to show.

The proof of transitivity now proceeds by a double induction. First, in the case when $p \equiv q \equiv \mathrm{ret}\ a$ and $v \equiv \mathrm{rret}\ a$, we have $v' : \mathrm{ret}\ a \doteq r$ and

$$\mathrm{trans}\ (\mathrm{ret}\ a)\ (\mathrm{ret}\ a)\ (\mathrm{rret}\ a)\ r\ v' \equiv v' : (\mathrm{ret}\ a) \doteq r.$$

The second case is where the four lemmata must be used — here we have to find a definition for

$$\mathrm{trans}\ (\mathrm{invk}\ c\ s)\ (\mathrm{invk}\ c\ t)\ (\mathrm{rinvk}\ h)\ r\ v' : \mathrm{invk}\ c\ s \doteq r.$$

This proof continues by program induction on r. The case when r is of ret-form is handled by lemma_1:

$$\mathrm{trans}\ (\mathrm{invk}\ c\ s)\ (\mathrm{invk}\ c\ t)\ (\mathrm{rinvk}\ h)\ (\mathrm{ret}\ b)\ v' \equiv \mathrm{abort}\ (\mathrm{lemma}_1\ d\ t\ b\ v') : \mathrm{invk}\ c\ s \doteq r.$$

The final and most difficult case is to define

$$\mathrm{trans}\ (\mathrm{invk}\ c\ s)\ (\mathrm{invk}\ c\ t)\ (\mathrm{rinvk}\ h)\ (\mathrm{invk}\ d\ u)\ v' : \mathrm{invk}\ c\ s \doteq \mathrm{invk}\ d\ u,$$

where $v' : \mathrm{invk}\ c\ t \doteq \mathrm{invk}\ d\ u$. First we make the abbreviation

$$l \equiv \mathrm{lemma}_2\ c\ t\ d\ u\ v' : c = d.$$

Next, note that

$$\mathrm{lemma}_4\ c\ s\ d\ u\ l\ m : \mathrm{invk}\ c\ s \doteq \mathrm{invk}\ d\ u$$

provided that

$$m : (x : w@c) \to s\ x \doteq u\ (\mathrm{subst}_R\ l\ x).$$

To find the value of m we use the outer induction hypothesis together with lemma_3. Assume that $x : w@c$, then

$$\mathrm{trans}\,(s\,x)\,(t\,x)\,(h\,x)\,(u\,(\mathrm{subst}_R\,l\,x))\,(\mathrm{lemma}_3\,c\,t\,d\,u\,v') : s\ x \doteq u\ (\mathrm{subst}_R\ l\ x).$$

Abstraction on x gives the definition of m.

Proof of Theorem 2

The notation $(w\ \Rightarrow)$ stands for the function of type $\mathrm{set} \to \mathrm{set}$ that takes a set A to the set $w \Rightarrow A$. As seen above, left identity holds up to definitional equality. Since program equality is reflexive, it holds up to program equality as well. That right identity holds up to program equality is demonstrated by induction on p. The case when $p \equiv \mathrm{ret}\ a$ is trivial. In case $p \equiv \mathrm{invk}\ c\ t$,

$$\mathrm{invk}\ c\ t \gg\!= \mathrm{ret} \equiv \mathrm{invk}\ c\ (\hat{x}\ t\ x \gg\!= \mathrm{ret}) : w \Rightarrow A,$$

and the result follows from 'rinvk' and the induction hypothesis. The proof of associativity is also by induction on p. Again, the case $p \equiv \mathrm{ret}\ a$ is trivial. In case $p \equiv \mathrm{invk}\ c\ t$, we compute

$$(\mathrm{invk}\ c\ t \gg\!= f) \gg\!= g \equiv \mathrm{invk}\ c\ (\hat{x}\ t\ x \gg\!= f) \gg\!= g \equiv \mathrm{invk}\ c\ (\hat{y}\ (t\ y \gg\!= f) \gg\!= g) : w \Rightarrow C,$$

and

$$\mathrm{invk}\ c\ t \gg\!= (\hat{x}\ f\ x \gg\!= g) \equiv \mathrm{invk}\ c\ (\hat{y}\ t\ y \gg\!= (\hat{x}\ f\ x \gg\!= g)) : w \Rightarrow C.$$

Again, the result follows from 'rinvk' and the induction hypothesis.

Finally, we must show that the bind operation is extensional with respect to program equality. Assume (α) that $p \doteq q$ and (β) that $f\ x \doteq g\ x$ for $x : A$. We must show that

$$p \gg\!= f \doteq q \gg\!= g.$$

The proof is by induction on the proof (α) of $p \doteq q$. In the first case, $p \equiv q \equiv \mathrm{ret}\ a$. Then $p \gg\!= f \equiv f\ a$ and $q \gg\!= g \equiv g\ a$, and $f\ a \doteq g\ a$ by (β). In the second case, $p \equiv \mathrm{invk}\ c\ s$ and $q \equiv \mathrm{invk}\ c\ t$. By the induction hypothesis,

$$s\ x \gg\!= f \doteq t\ x \gg\!= g,$$

for any x. But $p \gg\!= f \equiv \mathrm{invk}\ c\ (\hat{x}\ s\ x \gg\!= f)$ and $q \gg\!= g \equiv \mathrm{invk}\ c\ (\hat{x}\ t\ x \gg\!= g)$. The two right hand sides are equal by the induction hypothesis and 'rinvk'.

Proof of Lemma 4

What we need to prove is that the two ways of composing up the world maps in the diagram

$$w_1 \ \text{—}f\text{—}\circ\ w_2 \ \text{—}g\text{—}\circ\ w_3 \ \text{—}h\text{—}\circ\ w_4$$

are equal. By definition of equality between world maps, it suffices to show that, for any command c of the world w_4, the two programs $((f;g);h)\ c$ and $(f;(g;h))\ c$ are equal. The left hand side is equal to $(\mathrm{lift}\ (f;g)\ (h\ c))$ and the right hand side is equal to $(\mathrm{lift}\ f\ (\mathrm{lift}\ g\ (h\ c)))$. Thus, it suffices to show that $(\mathrm{lift}\ (f;g)\ p)$ and $(\mathrm{lift}\ f\ (\mathrm{lift}\ g\ p))$ are equal programs for any $p : w_3 \Rightarrow A$. In case $p \equiv \mathrm{ret}\ a$, we have

$$\mathrm{lift}\ (f;g)\ (\mathrm{ret}\ a) \equiv \mathrm{ret}\ a : w_1 \Rightarrow A$$

and

$$\mathrm{lift}\ f\ (\mathrm{lift}\ g\ (\mathrm{ret}\ a)) \equiv \mathrm{lift}\ f\ (\mathrm{ret}\ a) \equiv \mathrm{ret}\ a : w_1 \Rightarrow A,$$

for $a : A$. In case $p \equiv \mathrm{invk}\ c\ d : w_3 \Rightarrow A$, where $c : |w_3|$ and $d : w_3@c \to w_3 \Rightarrow A$, we have

$$\mathrm{lift}\ (f;g)\ (\mathrm{invk}\ c\ d) \equiv \mathrm{lift}\ f\ (g\ c) \gg\!= (\hat{x}\ \mathrm{lift}\ (f;g)\ (d\ x)) : w_1 \Rightarrow A,$$

and

$$\mathrm{lift}\ f\ (\mathrm{lift}\ g\ (\mathrm{invk}\ c\ d)) \equiv \mathrm{lift}\ f\ (g\ c \gg\!= (\hat{x}\ \mathrm{lift}\ g\ (d\ x))) : w_1 \Rightarrow A.$$

To show that these two programs are equal we use induction on $(g\ c)$. Thus, we prove that the proposition

$$\mathrm{lift}\ f\ m \gg\!= (\hat{x}\ \mathrm{lift}\ (f;g)\ (d\ x)) \doteq \mathrm{lift}\ f\ (m \gg\!= \hat{x}\ \mathrm{lift}\ g\ (d\ x)) : \mathrm{prop} \quad (\dagger)$$

is true for any $m : w_2 \Rightarrow A$, provided that the proposition we are about to prove holds for any $(d\ x)$, i.e., the proposition is proved under the assumption that the induction hypothesis

$$\text{lift}\ (f;g)\ (d\ x) \doteq \text{lift}\ f\ (\text{lift}\ g\ (d\ x)) : \text{prop} \qquad (\ddagger)$$

is true for all $x : w_3@c$. First $m \equiv \text{ret}\ a : w_2 \Rightarrow A$, for $a : A$,

$$\text{lift}\ f\ (\text{ret}\ a) \gg\!= \hat{x}\ \text{lift}\ (f;g)\ (d\ x) \equiv (\text{ret}\ a) \gg\!= \hat{x}\ \text{lift}\ (f;g)\ (d\ x) \equiv \text{lift}\ (f;g)\ (d\ a) : w_1 \Rightarrow A,$$

whereas

$$\text{lift}\ f\ ((\text{ret}\ a) \gg\!= \hat{x}\ \text{lift}\ g\ (d\ x)) \equiv \text{lift}\ f\ (\text{lift}\ g\ (d\ a)) : w_1 \Rightarrow A,$$

and the equality follows from the induction hypothesis ($\ddagger$). Next, let $m \equiv \text{invk}\ u\ v : w_2 \Rightarrow A$, and let L and R stand for the two sides that we must prove equal, i.e.,

$$\text{L} \equiv (\text{lift}\ f\ (\text{invk}\ u\ v)) \gg\!= \hat{x}\ \text{lift}\ (f;g)\ (d\ x) \equiv (f\ u \gg\!= \hat{y}\ \text{lift}\ f\ (v\ y)) \gg\!= \hat{z}\ \text{lift}\ (f;g)\ (d\ z) : w_1 \Rightarrow A,$$

and

$$\text{R} \equiv \text{lift}\ f\ ((\text{invk}\ u\ v) \gg\!= \hat{z}\ \text{lift}\ g\ (d\ z) \equiv \text{lift}\ f\ (\text{invk}\ u\ (\hat{y}\ (v\ y) \gg\!= \hat{z}\ \text{lift}\ g\ (d\ z))) \equiv f\ u \gg\!= (\hat{y}\ \text{lift}\ f\ ((v\ y) \gg\!= \hat{z}\ \text{lift}\ g\ (d\ z))) : w_1 \Rightarrow A.$$

Now recall the monad law of associativity, viz., that the proposition

$$(n \gg\!= \alpha) \gg\!= \beta \quad \doteq \quad n \gg\!= (\hat{y}\ \alpha\ y \gg\!= \beta) : \text{prop}$$

is true for any n, α, and β. Put $n \equiv f\ u$, $\alpha\ y \equiv \text{lift}\ f\ (v\ y)$, $\beta\ z \equiv \text{lift}\ (f;g)\ (d\ z)$, so that

$$\text{L} \equiv (n \gg\!= \alpha) \gg\!= \beta : w_1 \Rightarrow A$$

and

$$\text{M} \equiv n \gg\!= (\hat{y}\ \alpha\ y \gg\!= \beta) : w_1 \Rightarrow A.$$

are equal. To complete the proof, we need to show that M and R are equal and appeal to transitivity of program equality. Substituting the definitions of n, α, and β into M gives

$$\text{M} \equiv f\ u \gg\!= (\hat{y}\ \text{lift}\ f\ (v\ y) \gg\!= \hat{z}\ \text{lift}\ (f;g)\ (d\ z)) : w_1 \Rightarrow A.$$

By comparing with R, we see that we require extensionality of bind and the equality

$$\text{lift}\ f\ (v\ y) \gg\!= \hat{z}\ \text{lift}\ (f;g)\ (d\ z) \doteq \text{lift}\ f\ (v\ y \gg\!= \hat{z}\ \text{lift}\ g\ (d\ z)),$$

but this is just the induction hypothesis of the induction on m, i.e., (†) with $m \equiv v\ y$.

Proof of Theorem 7

There are several things to establish. First, that the above mapping defines a functor from Wm to $\mathrm{Mnd}^{\mathrm{op}}$. The proof that id_w is an identity with respect to composition of world maps also shows that lift id_w is equal to the identity function. The proof that composition of world maps is associative shows that lift $(f;g) \cong (\text{lift}\ f) \circ (\text{lift}\ g)$, where $(\cong)$ denotes equality of monad morphisms. To establish that we have a functor, one more thing is required, viz., that $(\text{lift}\ m)$ indeed is a monad morphism, i.e., is that the proposition

$$\text{lift}_B\ m\ (n \gg\!= f) \doteq \text{lift}_A\ m\ n \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) : \text{prop}$$

is true, for any $n : w_2 \Rightarrow A$ and $f : A \to w_2 \Rightarrow B$. The proof is by induction on n. The case when $n \equiv \text{ret}\ a$ is proved by

$$\text{lift}_B\ m\ (\text{ret}\ a \gg\!= f) \equiv \text{lift}_B\ m\ (f\ a)$$

and

$$\text{lift}_A\ m\ (\text{ret}\ a) \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) \equiv \text{ret}\ a \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) \equiv \text{lift}_B\ m\ (f\ a).$$

Let $n \equiv \text{invk}\ c\ t$. The induction hypothesis is that the proposition

$$\text{lift}_B\ m\ (t\ x \gg\!= f) \doteq \text{lift}_A\ m\ (t\ x) \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) : \text{prop} \qquad (\dagger)$$

is true for any $x : w_2@c$. We compute:

$$\begin{aligned} &\text{lift}_B\ m\ (\text{invk}\ c\ t \gg\!= f) \\ &\equiv \text{lift}_B\ m\ (\text{invk}\ c \gg\!= (\hat{x}\ t\ x \gg\!= f)) \\ &\equiv m\ c \gg\!= (\hat{x}\ \text{lift}_B\ m\ (t\ x \gg\!= f)) \\ (\dagger) &\doteq m\ c \gg\!= (\hat{x}\ \text{lift}_A\ m\ (t\ x) \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x))), \end{aligned}$$

and

$$\begin{aligned} &\text{lift}_A\ m\ (\text{invk}\ c\ t) \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) \\ &\equiv (m\ c \gg\!= (\hat{x}\ \text{lift}_A\ m\ (t\ x))) \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x)) \\ (\text{assoc.}) &\doteq m\ c \gg\!= (\hat{x}\ \text{lift}_A\ m\ (t\ x) \gg\!= (\hat{x}\ \text{lift}_B\ m\ (f\ x))), \end{aligned}$$

and the two right hand sides are equal.

To show the functor is *full*, i.e., that any morphism between world-based monads arises from a world map by lifting, assume that w_1 and w_2 are worlds, and that we have monad morphism

$$f_A : (w_2 \Rightarrow A) \to (w_1 \Rightarrow A),$$

with

$$f_A\ (\text{ret}\ a) \doteq (\text{ret}\ a), \text{ and}$$
$$f_B\ (a \gg\!= g) \doteq f_A\ a \gg\!= (\hat{x}\ f_B\ (g\ x)).$$

If $c : |w_2|$ and $t : w_2@c \to w_1 \Rightarrow B$, we have

$$f_B\ (\text{invk}\ c\ t) \equiv f_B\ (\text{invk}\ c\ \text{ret} \gg\!= t) \doteq f_{w_2@c}\ (\text{invk}\ c\ \text{ret}) \gg\!= (\hat{x}\ f_B\ (t\ x)).$$

If we make the definition $m\ c \equiv f_{w_2@c}\ (\text{invk}\ c\ \text{ret})$, the definition of 'lift' gives extensional equality between f_B and $\text{lift}_B\ m$ as $\text{lift}_A\ m\ (\text{ret}\ a) \equiv \text{ret}\ a$, and

$$\text{lift}_B\ m\ (\text{invk}\ c\ t) \equiv m\ c \gg\!= (\hat{x}\ \text{lift}_B\ m\ (t\ x)).$$

That is f satisfies the equations that define lift m, but up to program equality instead of definitional equality.

To show that the functor is *faithful*, assume that lift $m \cong$ lift n, for world maps $m, n : w_1 \multimap w_2$. For any $c : |w_2|$, we have lift m (invk c ret) $\equiv m\ c \gg\!= \text{ret}$, and similarly for n. It follows by right associativity that $m\ c \doteq n\ c$, as required.

REFERENCES

[1] M. Abbott, T. Altenkirch, and N. Ghani. "Categories of Containers". In: *Foundations of Software Science and Computation Structures*. Vol. 2620. Lect. Notes Comput. Sc. 2003, pp. 23–38.

[2] M. Abbott, T. Altenkirch, and N. Ghani. "Containers: Constructing strictly positive types". In: *Theor. Comput. Sci.* 342 (2005), pp. 3–27.

[3] J. W. Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". In: *Communications of the ACM* 21.8 (1978), pp. 613–641.

[4] L. M. D. C. S. Barbosa. "Components as Coalgebras". PhD thesis. Departamento de Informática, Escola de Engenharia, Universidade do Minho, 2001.

[5] A. Buisse and P. Dybjer. "The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories – an Intuitionistic Perspective". In: *Electron. Notes Theor. Comput. Sci.* 218 (2008), pp. 21–32.

[6] P. Dybjer. "Inductive families". In: *Formal Aspects of Computing* 6 (1994), pp. 440–465.

[7] D. Garlan and M. Shaw. *An Introduction to Software Architecture*. Tech. rep. CMU-CS-94-166. School of Computer Science, Carnegie Mellon University, 1994.

[8] N. Ghani and T. Uustalu. "Coproducts of ideal monads". In: *Rairo Inf. Theor. Appl.* 38.4 (2004), pp. 321–342.

[9] J. A. Goguen. "Categorical foundations for general systems theory". In: *Proceedings of the European meeting, Vienna*. Ed. by F. Pichler and R. Trappl. Advances in Cybernetics and Systems Research. Transcripta Books, 1972, pp. 121–130.

[10] P. Hancock and A. Setzer. "Interactive programs in dependent type theory". In: *Computer Science Logic*. Vol. 1862. Lect. Notes Comput. Sc. 2000, pp. 317–331.

[11] P. Hoogendijk and O. de Moor. "Container types categorically". In: *J. Funct. Programming* 10.2 (2000), pp. 191–225.

[12] P. Hudak et al. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–55.

[13] H. C. Lauer and R. M. Needham. "On the Duality of Operating System Structures". In: *Proc. Second International Symposium on Operating Systems*. 1978.

[14] C. Lüth and N. Ghani. "Composing monads using coproducts". In: *ICFP*. 2002, pp. 133–144.

[15] P. Martin-Löf. "Constructive mathematics and computer programming". In: *Logic, Methodology and Philosophy of Science VI*. Ed. by L. J. Cohen et al. Amsterdam: North-Holland, 1982, pp. 153–175.

[16] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Napoli: Bibliopolis, 1984.

[17] E. Moggi. "Computational lambda-calculus and monads". In: *Proceedings of the Fourth Annual Symposium on Logic in computer science*. Piscataway, NJ, USA: IEEE Press, 1989, pp. 14–23.

[18] B. Russell. "Mathematical Logic as Based on the Theory of Types". In: *Amer. J. Math.* 30.3 (1908), pp. 222–262.

[19] W. Swierstra. "Data types à la carte". In: *J. Funct. Programming* 18.4 (2008), pp. 423–436.

[20] D. A. Turner. "Elementary strong functional programming". In: *First International Symposium on Functional Programming Languages in Education*. Ed. by R. Plasmeijer and P. Hartel. Vol. 1022. 1995, pp. 1–13.