Categorising Non-interference

Jeremy Jacob
Department of Computer Science
The University of York, York, YO10 5DD
Jeremy.Jacob@cs.york.ac.uk; http://www-users.cs.york.ac.uk/~jeremy/

Sunday 8 April 1990

Abstract

Non-interference [Goguen & Meseguer 1982] is given an abstract definition in category-theoretic terms. Unwinding theorems are investigated from this starting point. We conclude that category theory is a powerful tool for reasoning about non-interference.

1. Introduction

The notion of non-interference was introduced in [Goguen & Meseguer 1982]. It is an information-flow security property of systems. If a user $U$ is not able to make changes to a system $S$ which are detectable by another user $V$ we say $U$ is noninterfering with $V$ across $S$. We will refer to $U$ as the acting user and $V$ as the viewing user.

It is often difficult to assess a system for non-interference directly from the definition as the formal definition is in terms of all histories of the system. In order to ease the burden of demonstrating that a system is non-interfering various theorems, with the general title of Unwinding Theorems, have been demonstrated. The purpose of these theorems is to give conditions on each state-operation pair which imply non-interference (see, for example, [Goguen & Meseguer 1984, Fine et al 1989].)

The purpose of this paper is to give a very abstract definition of non-interference and establish a framework in which we can prove general unwinding theorems. The framework in which we work is one which encourages and aids such generality: category theory. Thus our work is in the spirit of [Goguen 1989]. Category theory allows our proofs to proceed mostly by simple equational reasoning; it is the simplicity of these proofs which partly justifies this work.
Notation

A brief introduction to category theory—sufficient only for the purposes of this paper—is given in appendix A. As we are using \( f; g \) to mean “\( f, \) then \( g \)” we use a style where we place the argument to a function before the function. Thus we write \( v \) transformed by \( f \):

\[
v \triangleright f
\]

instead of \( f \) applied to \( v \):

\[
f(v)
\]

An advantage of this decision is the simple form of the pseudo-associative law:

\[
v \triangleright (f; g) = (v \triangleright f) \triangleright g
\]

2. Non-Interference

We will first give an abstract definition of non-interference in category theoretic terms, and then justify it.

Definition 1 [Non-interference] Let \( p \) and \( v \) be arrows in a category such that \( \bar{p} = \bar{p} = \bar{v} = i \), say. We say that \( p \) is non-interfering with \( v \), written \( p \not\triangleright v \), if and only if \( v \) co-equalises \( p \) and \( i \). Figure 2 illustrates this definition in the usual category theory manner.
The first thing to notice is that this has a different structure from the informal definition of non-interference given in section 1: definition 1 has two free variables \((p \text{ and } v)\) whereas the informal definition has three \((U, V \text{ and } S)\). The usual way of expressing non-interference is to require the viewing user’s output after the machine has executed a sequence of instructions to be the same as that after the machine has executed the same sequence of instructions, but purged of the acting user’s instructions. We have wrapped the details of the machine and the viewing user together, as \(v\), to get an arrow that represents determining the viewing user’s output after a sequence of instructions. Purging of the acting user’s instructions is represented by \(p\). Now, writing out definition 1 we get:

\[
p \not\rightarrow v \iff v = i; v = p; v
\]

This expresses the fact that determining the viewer’s output is independent of the acting user’s behaviour. If \(v\) happens to be the co-equaliser of \(p\) and \(i\) then it represents the most general (group of) user(s) that is not interfered with by the user represented by \(p\).

So far we have talked of \(p\) and \(v\) representing a user or a user’s view of a machine. We prefer to think of \(p\) and \(v\) defining these concepts. By exhibiting an arrow with source and target equal we are defining an acting user.

### 3. A simple case

Our first unwinding theorem is particularly simple. The category within which we work is the category of categories \(\text{CC}\) (theorem 5). The arrows in this category are functors.

**Theorem 1** Let \(C\) be a category and \(G\) be a set of generating arrows for the category. Let \(p : C \to C\). Let \(v\) be a functor with source \(C\) such that the unwinding condition:

\[
\forall g : G \cdot g \triangleright (p; v) = g \triangleright v
\]

holds. Then \(p \not\rightarrow v\).

**Proof** By induction on the number of generators that represent an arrow.

**Base case** Direct from the unwinding condition.

**Inductive step** Suppose \(a\) is an arrow representable by \(n + 1\) generators, say \(a = g; b\) where \(b\) is representable by \(n\) generators and \(g \in G\), and let \(b \triangleright (p; v) = b \triangleright v\). Then

\[
(g; b) \triangleright (p; v) = (g \triangleright (p; v)); (b \triangleright (p; v)) = (g \triangleright v); (b \triangleright v) = (g; b) \triangleright v
\]

The second equality depends on both the unwinding condition and the induction hypothesis, the others follow from laws about functors. Note that the composition operator, \(\cdot\), in these equations is overloaded with three meanings: composition of functors, composition in \(C\) and composition in \(\text{v}\).
The unwinding condition of theorem 1 says that—if \( p \) and \( v \) are functors—all we need to do to show non-interference is to check that each generator has the same effect on the viewing users output as does the purged form of the generator.

When \( g \triangleright p = g \) (and so the command is one in which there is no contribution from the acting user) there is nothing to check. This allows us to rewrite the unwinding condition:

\[
\forall g : G \cdot g \triangleright p \neq g \Rightarrow g \triangleright (p;v) = g \triangleright v
\]

The problem with theorem 1 is that \( v \) will not often be a functor. If \( v \) is a functor it means that, as far as the viewing user is concerned, there is only a single state. The reason is as follows: Each generator represents a simple command which can be issued to the machine, and we include the null command \texttt{skip} among these. Then composition ‘;’ in \( C \) is exactly sequential composition of ordinary programming languages. If \( v \) is a functor, then

\[
(s;t) \triangleright v = (s \triangleright v) ; (t \triangleright v)
\]

that is, the output of a sequence of commands \( t \) is independent of the sequence of commands that precede it. This is equivalent to saying that the states achieved by two sequences of commands are equivalent as far as \( v \) is concerned.

Example 1 A simple pocket calculator (that is an expression evaluator with no store for partial results), provides its client with a one-state machine. If the calculator is implemented as the only facility available to a user \( V \) on a shared machine, theorem 1 can be applied to show non-interference by other users with the calculator user.

Insisting that \( p \) is a functor means that whether or not a command is effected by \( p \) is independent of when it occurs. If we want to model mutating security levels we cannot have \( p \) as a functor.

Example 2 Let \( C \) be a monoid of commands (so any sequence of commands is allowed). Suppose there is a user called \( a \), and all of that user’s actions are labelled with its name, for example \( a.c \). Normally all of \( a \)’s actions are public except between the commands \( p \) (for “private”) and \( o \) (for “open”). Let \( c_i \), for \( i \in \mathbb{N} \), be arbitrary commands that do not change any security levels and let \( b \) be a user different from \( a \). Let \( g, s \in Arw(C) \) and \( g \) be a single command. Then a non-functoral purge function is defined:

\[
\begin{align*}
\langle \rangle \triangleright q & = \langle \rangle \\
(g;s) \triangleright q & = g;(s \triangleright q) \quad \text{if } g \neq a.p \\
& = s \triangleright q' \quad \text{otherwise}
\end{align*}
\]

where

\[
\begin{align*}
\langle \rangle \triangleright q' & = \langle \rangle \\
(g;s) \triangleright q' & = g;(s \triangleright q') \quad \text{if } g \triangleright \text{user} \neq a \\
& = s \triangleright q' \quad \text{if } g \triangleright \text{user} = a \land g \neq a.o \\
& = s \triangleright q \quad \text{otherwise}
\end{align*}
\]

\[
(x.d) \triangleright \text{user} = x
\]
Then, for example:
\[(a.c_1; b.c_2; a.p; a.c_3; a.o; a.c_4) \triangleright q = a.c_1; b.c_2; a.c_4\]

and
\[(a.c_3; a.o; a.c_4) \triangleright q = (a.c_3; a.o; a.c_4)\]

The purge function needs to know which actions are hidden and which public. To do this it cannot use purely local information about a command; it needs to know about the earlier actions. Functors may use local information only.

In section 5 we relax both the conditions that \(p\) and \(v\) are a functors. Before we do that we consider how we commonly interpret a theorem like theorem 1.

### 4. Interpretation

The normal model of a system is as a monoid, as in example 2. A minimal set of generators of this category are the individual commands of a machine, including the null command. That the category is a monoid means that any sequence of commands is possible. A syntactically illegal sequence of commands is modelled by having its output be an error message. Henceforth we will restrict ourselves to monoids.

There are two interesting ways of particularising the category further, which are two ends of a spectrum. The benefit of category theory is that our theorems apply to the whole spectrum.

#### 4.1. Interleaved commands

In this case each generator is contributed to by exactly one user. Each generator represents a user/instruction pair and \(p\) has the property:

\[
\forall g : G \cdot g \triangleright p = i \lor g \triangleright p = g
\]

where \(i\) is the unique object in the monoid. This is the computational model in which non-interference and similar properties (for example “composability” [McCullough 1988] and “restriction of information-flow” [Jacob 1988]) are usually defined.

In this case \(p\) picks out the acting user’s commands as those generators \(g\) for which:

\[g \triangleright p = i\]

\(p\) has no effect on any of the other commands; that is:

\[g \triangleright p = g\]

if \(g\) is not a command issued by the acting user.
4.2. Concurrent commands

At this end of the spectrum each generator is contributed to by every user. This is the case of clocked systems, where each user must contribute a command on each clock pulse; the command may be the null command. An unwinding theorem for such a system is considered in [Fergusson 1989]. We can consider the monoid as being the limit (“limit” is the categorical generalisation of cartesian product) of a family of monoids, one for each user. Let \( \pi_j \) be the projection function from the limit to the \( j^{th} \) member of the family, then:

\[
g \triangleright (p; \pi_j) = i \triangleright \pi_j
\]

if \( j \) is the index of the member of the family which corresponds to the acting user and \( i \) is the object in this monoid. For \( k \neq j \) we have:

\[
g \triangleright (p; \pi_k) = g \triangleright \pi_k
\]

5. The general case

From now on we will work with the category of sets, \( SS \), (theorem 7). We do this so that the arrows \( p \) and \( v \) need not be functors, but merely total functions. We do not proceed in complete generality, however, but make some fairly unrestrictive assumptions about \( p \) and \( v \).

5.1. Structuring the viewing arrow

The technique of denotational semantics recommends that the function, \( m \), which associates a meaning (usually as a state-to-state function) with a program is a functor [Stoy 1977]. That is, it is required that

\[
x \triangleright ([\text{skip}] \triangleright m) = x
\]

\[
[p; q] \triangleright m = ([p] \triangleright m); ([q] \triangleright m)
\]

The first of these equations just says that \( m \) preserves the identity and the second that the meaning of a sequential composition can be determined from the meaning of its components. Now \( \tilde{m} \) is a category of endofunctors over a category of states, \( S \) say. So \( m : C \rightarrow [S \rightarrow S] \), where \( C \) is a monoid (of commands).

We can take advantage of this insight to split \( v \) into two parts: a semantic functor \( m \) and a local view functor \( w \). We will also need an initial state, \( s \). To summarise: we are interested only in \( v \) which can be expressed in the form:

\[
v = (s \triangleright m'); w
\]

where \( m' \) is the ‘next state function’ defined by

\[
a \triangleright (s \triangleright m') = s \triangleright (a \triangleright m)
\]
(Note that \( m' \) is not a functor, or arrow in the category of categories, but is an arrow in the
category of sets.) We can rewrite \( t \triangleright v \) by

\[
t \triangleright v = s \triangleright ((t \triangleright m); w)
\]

We can go further, and identify \( C \) with a submonoid of the monoid of functors over \( S, [S \rightarrow S] \). Thus all sequences of commands which have the same effect are identified. In this
case \( m \) is the identity functor on \( C \), and we express \( v \) in the form:

\[
v = s \triangleright (t; w)
\]

Note that if the target of \( w \) is a monoid (that is, there is only one state from the viewing
user’s point of view) then \( v \) is a functor.

5.2. Structuring the acting arrow

The arrow \( p \) may use information about the past to decide whether or not a command can be
purged; it may not use information about the future. That is, \( p \) must have the form:

\[
(t; u) \triangleright p = (t \triangleright p); (u \triangleright q_t)
\]

The function \( q_t \) tells us how to transform \( u \) in the state reached by doing \( t \). Note that if

\[
\forall s : Arw(C) \cdot q_s = p \text{ then } p \text{ is a functor.}
\]

If the viewing arrow is structured as \( (s \triangleright m'); w \) then we require that \( m \) and \( q \) must satisfy

\[
\forall t, u : Arw(C) \cdot t \triangleright m = u \triangleright m \implies q_t = q_u
\]

This says that if two executions achieve the same state then the rules for purging after them
must be the same. We will take advantage of this to write \( q_{s'} \), where \( s' \) is a state, rather than
\( q_t \), when \( s' = s \triangleright (t \triangleright m) \) and \( s \) is the initial state of the system.

5.3. Unwinding

In order to prove an unwinding theorem we need unwinding conditions. We prove a theorem
below with two conditions. For a category \( S, C \) a subcategory of \( [S \rightarrow S] \), a set of generators
\( G \) of \( C \), \( qa (a \in Obj(S)) \) structured as above and any functor \( w \) with source \( S \), they are:

\textbf{VS} To calculate the viewing user’s new state each command uses only that part of the state
visible to the viewing user.

\[
\forall a, b : Obj(S), g : G \cdot a \triangleright w = b \triangleright w \Rightarrow a \triangleright (g; w) = b \triangleright (g; w)
\]

It is trivial to prove by induction that \textbf{VS} implies:

\[
\forall a, b : Obj(S), t : Arw(C).
\]

\[
a \triangleright w = b \triangleright w \Rightarrow a \triangleright (t; w) = b \triangleright (t; w)
\]
VC The acting user’s inputs to a command have no effect on the viewing user’s new state
\[ \forall a : \text{Obj}(S) \cdot (g \triangleright q_a); w = g; w \]

Now we can easily prove a theorem which generalises VC:

**Theorem 2** Let \( S \) be some category and \( G \) be a set of generators for a subcategory \( C \) of \( [S \to S] \). Let \( s \in \text{Obj}(S) \). For each \( a \in \text{Obj}(S) \) let \( q_a \) be a function from \( \text{Arw}(C) \) to \( \text{Arw}(C) \) with the property:
\[ \forall t : C, g : G \cdot (t; g) \triangleright q_a = (t \triangleright q_a); (g \triangleright q_{awt}) \]
(Note that this implies \( p = q_s \).)
Let \( w \) be a functor with source \( S \). Further, suppose the unwinding conditions \( \text{VS} \) and \( \text{VC} \) hold. Then
\[ \forall a : \text{Obj}(S), t : \text{Arw}(S) \cdot ((t \triangleright q_a); w) = (t; w) \]

**Proof** By induction over the number of generators that represent an arrow in \( C \).

**Base case** Directly from VC.

**Induction step** Let \( u \in \text{Obj}(C) \) be representable by \( n + 1 \) generators, say \( u = t; g \) where \( t \) is representable by \( n \) generators and \( g \in G \). Let \( b \in \text{Obj}(S) \). Suppose the theorem holds for \( t \). Then
\[ b \triangleright ((t \triangleright q_a); w) = b \triangleright (t; w) \]
\[ \iff [\text{Defn } \triangleright, S \to S] \]
\[ (b \triangleright (t \triangleright q_a)) \triangleright w = (b \triangleright t) \triangleright w \]
\[ \implies [\text{VS}] \]
\[ (b \triangleright (t \triangleright q_a)) \triangleright (g; w) = (b \triangleright t) \triangleright (g; w) \]
\[ \iff [\text{VC}] \]
\[ (b \triangleright (t \triangleright q_a)) \triangleright (g \triangleright q_a) = (b \triangleright t) \triangleright (g; w) \]
\[ \iff [\text{Defn } \triangleright, S \to S, q_c] \]
\[ b \triangleright (((t; g) \triangleright q_a); w) = b \triangleright ((t; g); w) \]

\[ \square \]

Now it is simple to establish

**Theorem 3** Under the conditions of theorem 2, and with \( s \in \text{Obj}(S) \), \( v \) defined by:
\[ \forall t : C \cdot t \triangleright v = s \triangleright (t; w) \]
and \( p \) defined by:
\[ p = q_s \]
we have \( p \not\triangleright v \).

**Proof**
\[ t \triangleright (p; v) \]
\[ = [\text{Defn } p, v] \]
\[ s \triangleright ((t \triangleright q_s); w) \]
\[ = [\text{Theorem 2, } a = s] \]
\[ s \triangleright (t; w) \]
\[ = [\text{Defn } v] \]
\[ t \triangleright v \]
6. Many users

Usually there is more than one acting and one viewing user of a system, and it is usual to impose a security classification on these users. Now we extend our definitions to this case.

The user classification is normally discussed in terms of a partial order over disjoint sets of users but it is more convenient to discuss the classification in terms of a pre-order (definition 7) over users. Given a pre-order the partial order over disjoint sets can be easily constructed; also, given a partial order over disjoint sets, the pre-order can be generated. The details are left to the reader.

Let \( U \) be a pre-order of user names, where \( x \geq y \) means information may flow from \( y \) to \( x \), and let \( N \) be the category in which we will try and find non-interfering pairs of arrows. We need two functions from \( U \) to \( N \), one, \( P \) say, to give the arrow representing each user’s acting behaviour and one, \( V \) say, to give the arrow representing each user’s viewing behaviour.

Now we can say that \( P \) and \( V \) enforce the information-flow constraints implied in \( U \):

**Definition 2** [Many-user non-interference] Let \( U \) be a pre-order, \( N \) be any category and \( P \) and \( V \) be functions \( \text{Obj}(U) \rightarrow \text{Arw}(N) \), with the properties:

\[
\forall x : \text{Obj}(U) \cdot x \triangleright P = x \triangleright P
\]

\[
\forall x, y : \text{Obj}(U) \cdot x \nRightarrow y \Rightarrow x \triangleright P = y \triangleright V
\]

Then \( P \) and \( V \) enforce the information-flow constraints of \( U \) if:

\[
\forall x, y : \text{Obj}(U) \cdot x \nRightarrow y \Rightarrow x \triangleright P \nRightarrow y \triangleright V
\]

We record this \( P \nRightarrow_U V \).

The unwinding theorems for this case are the obvious generalisations of the theorems of sections 3 and 5 which say we need to check every pair of users for which \( \nRightarrow \) holds.

7. Conclusions

Our conclusions are of two kinds: on the method we have used, and on the subject matter treated.

Our method is category theory. The formulation of non-interference (definition 1) is as abstract as can be and category theory directly helped to achieve this. The theorems almost proved themselves; the proof method of category theory—equational reasoning—is a very simple, yet very powerful technique; it is therefore a very enjoyable method to use. The author of this paper is able to join Goguen in recommending the categorical frame of mind for investigating topics in computing science [Goguen 1989].
We have shown an unwinding theorem (theorem 3) that applies to a very broad spectrum of types of system. It generalises the unwinding theorems of [Goguen & Meseguer 1984] and [Fergusson 1989], for example. Our theorems assume that commands form a monoid. We have not been as abstract as we could, by dealing with monoids of commands rather than categories whose arrows are commands. Thus our results do not apply to systems where some sequences of commands are syntactically invalid. The extension to categories is a tedious task (it is necessary to insert conditions about matching sources and targets at many points and we may not identify commands with the monoid of functors over states) but should be straightforward; carrying out this task would generalise these results to languages where not every string is a syntactically valid program.

The unwinding condition $\mathbf{VC}$ is probably stronger than necessary, as it requires an equality for all pairs of states. However, not all such pairs may arise. Hence there is some scope for weakening the condition.

Acknowledgements

This work was funded through the SCEIP unit of the RSRE. In particular I am grateful to Ruaridh Macdonald and Colin O’Halloran of that unit. I would also like to thank Joseph Goguen for encouraging me to improve this paper by daring to be abstract.

References

[Fergusson 1989]

[Fine et al 1989]

[Goguen 1989]

[Goguen & Meseguer 1982]

[Goguen & Meseguer 1984]
Appendix

A. Definitions and terms from category theory

In just the same way that group theory abstracts the common features of addition, multiplication, etc., category theory abstracts the essential features of functional composition. The basic components of a category are objects (which abstract the elements to which functions can be applied and the results of function application), arrows (which abstract functions) and composition of arrows (which abstracts functional composition). Categories also come with source and target operators which relate arrows to objects (these operators abstract the domain and range operators). The essential properties of a category are the associativity of the composition operator and the existence of identity arrows for each object. We can use the identity arrows to stand for the objects they are identities on; this gives us the following formalisation of a category:

Definition 3 [Category] A collection of arrows, $A$, together with a binary composition operator on arrows, “$\cdot$”, and unary source and target operators $\cdot\leftarrow$ and $\cdot\rightarrow$ is a category if, for arrows $a$, $b$ and $c$:

$$(a;b)\text{ is defined } \iff a = b$$

$$\begin{align*}
\overrightarrow{a} \cdot \overrightarrow{a} &= \overrightarrow{a} \\
\overrightarrow{a} &= \overrightarrow{a} \\
\overrightarrow{a} &= \overrightarrow{a} \\
a &= a \\
a &= a \\
(a;b) &= a \\
(a;b) &= b \\
(a;b);c &= a;(b;c)
\end{align*}$$

We will write $\text{Arw}(C)$ for the arrows of a category $C$. \qed
This list of axioms is not as formidable as it looks. Most are to establish the relationship between the arrow which is produced as a result of \( \rightarrow \) (or \( \leftarrow \)) and the properties of identity arrows. We can recover the objects of a category from the identity arrows.

**Definition 4** [Objects] The objects of a category are the arrows \( a \) for which:

\[
\bar{a} = a = \bar{a}
\]

We write \( \text{Obj}(C) \) for the set of all such arrows in category \( C \).

A set of generators of a category is a subset of the arrows of a category from which all other arrows can be fabricated:

**Definition 5** [Generator] Let \( C \) be a category, and let \( G \) be a subset of \( \text{Arw}(C) \). \( G \) generates \( C \) if every arrow in \( C \) can be expressed in the form \( g_0; \ldots; g_n \) for some finite number of arrows \( g_0, \ldots, g_n \in G \).

We now consider two special types of categories. The first is a category with only one object.

**Definition 6** [Monoid] A category \( M \) is a monoid if:

\[
\#\text{Obj}(M) = 1
\]

Another special case is a category in which there is at most one arrow from one object to another. (Note: given two objects, there can be an arrow in each direction.)

**Definition 7** [Pre-order] A category \( C \) is a pre-order if, for all arrows of the category \( a \) and \( b \),:

\[
\bar{a} = \bar{b} \land \bar{a} = \bar{b} \Rightarrow a = b
\]

We record the existence of an arrow from object \( p \) to object \( q \) by \( p \leq q \).

A functor is a structure preserving map between categories.

**Definition 8** [Functor] A map \( F \) from the arrows of one category \( C \) to the arrows of another \( D \) is a functor if, for any arrow \( c \) of \( C \):

\[
\begin{align*}
\bar{c} \circ F &= \bar{c} \circ F \\
\bar{c} \circ F &= \bar{c} \circ F
\end{align*}
\]

If \( b \) is also an arrow of \( C \) such that \( \bar{b} = \bar{c} \) we require:

\[
(b; c) \circ F = (b \circ F); (c \circ F)
\]

We write \( F : C \rightarrow D \) if \( F \) is a functor from \( C \) to \( D \).
Note that the \( -, \vdash, \Rightarrow \) operators on the left hand sides of the equations in definition 8 are those of \( C \), while those on the right hand side are those of \( D \).

**Definition 9** [Endofunctor] A functor \( F : A \to B \) is an endofunctor if and only if \( A = B \) \( \square \)

A functor on a pre-order is a monotonic function.

Every category \( C \) comes with an identity (endo)functor \( I_C \):

**Theorem 4** For any category \( C \), define \( I_C \) by:

\[
\forall a \in \text{Arw}(C). \quad a \vdash I_C = a
\]

Then \( I_C : C \to C \).

**Proof** Trivial, from definition 8. \( \square \)

An important special case of a category is the category of categories:

**Theorem 5** Let \( CC \) have as collection of arrows all functors. Define \( \vdash, \Rightarrow \) by:

\[
(F : C \to D) \vdash I_C \quad \text{and} \quad (F : C \to D) \Rightarrow I_D
\]

Define composition of functors \( F : C \to D \) and \( G : D \to E \) by

\[
c \Rightarrow (F; G) = (c \Rightarrow F) \Rightarrow G
\]

Then \( CC \) is a category.

**Proof** This is a well known result of category theory, easily checked from definition 3. \( \square \)

The objects of \( CC \) are formally the identity functors and informally the categories themselves.

An important class of categories are the monoids of endofunctors.

**Theorem 6** Let \( C \) be a category, and let \( [C \to C] \) have as collection of arrows all endofunctors of \( C \). When \( \vdash, \Rightarrow \) and \( \vdash \) are inherited from \( CC \), \( [C \to C] \) forms a monoid with \( \text{Obj}([C \to C]) = \{ I_C \} \).

**Proof** This is a well known result of category theory. \( \square \)

Another important category is the category of sets:

**Theorem 7** Let \( SS \) have as collection of arrows all total functions on sets. Define \( \vdash \) and \( \Rightarrow \) as domain and range respectively. \( \vdash \) is defined as ordinary functional composition. Then \( SS \) is a category.

**Proof** This is a well known result of category theory, easily checked from definition 3. \( \square \)

Lastly, we define the notion of a co-equaliser.
**Definition 10** [Co-equaliser] Let $f$ and $g$ be two arrows in a category $C$. The arrow $e$ is the co-equaliser of $f$ and $g$ if and only if

$$f; e = g; e$$

and,

$$\forall h: Arw(C) \cdot \exists k: Arw(C) \cdot f; h = g; h \Rightarrow h = e; k$$

If an arrow only has the first of these properties we will just say that it co-equalises $f$ and $g$. This definition is pictured in figure 1.

The second clause of the definition of a co-equaliser says that it is, in some sense, the smallest such arrow.