Atomic Functions

by Sven Nilsen, 2019

In this paper I explain what is meant by an “atomic function” in path semantics.

An atomic function is a function that takes a unique input and returns a unique output.
When there are multiple interpretations of “unique”, the uniqueness of atomic functions is the lowest.
The level of uniqueness equal to atomic functions, might be referred to as “atomically unique”.

For example, when increasing a number \`0\` by one, it returns \`1\`:

inc(0) = 1

This is an atomic function.
The function \`inc\` is **not** the atomic function.
The whole expression \`inc(0) = 1\` is the atomic function.

Neither is \`inc(0)\` an atomic function, yet it is atomically unique.
The atomic function can be constructed in a context where \`inc(0)\` is defined as a unique value.
Since the atomic function is reachable by construction it this context, it has the same uniqueness level.

Atomic functions have multiple parts, and are only considered atomic functions as a whole.

The function \`inc\` is not atomically unique, but it is atomically unique for the input \`1\`:

inc(1) = 2

An atomic function can also take multiple arguments:

and(true, true) = true

All atomic functions consist of 3 parts:

1. The identifying part (usually a symbol expressing some idea or definition)
2. The input
3. The output

For every atomic function, there exists an associated atomic sub-type:

<input> : [<identifier>] <output>

The sub-type is called “atomic” because it has the same uniqueness level as atomic functions.

When one starts with some input and constructs an atomic sub-type from it, it is called “taking a path”.
Paths are the building blocks of the semantics of functions, therefore the name “path semantics”.

For example:

and(true, true) = true

Has an associated atomic sub-type:

(true, true) : [and] true

Here, `[and] true` “takes the path” from `(true, true)`.

There is no formal definition of what “takes the path” can mean as a whole.
It depends on the context where it is done and usage of the language.
Therefore, there can be multiple kinds of paths within the same language, or similar kinds of paths between languages that uses different definitions.

Usually, atomic functions are used to explain how path semantics works at type level:

true(bool) → true	false(bool) → false
bool : [true] true	bool : [false] false

All members are atomic functions taking the type as argument and returning themselves.
The member identifier is atomically unique.

Since the member identifier is atomically unique, one can infer it from the output value:

bool : [:] true	bool : [:] false

The symbol `[:]` means that it can be inferred from context and it has a nice symmetry:

true : bool	<~=>	bool : [:] true
false : bool	<~=>	bool : [:] false

Here, `<~=>` means equivalent. The left side can be deduced from the right side and vice versa.

Using tuples, one can also define members of a type as a set:

bool : [:] (true, false)   <~=>   bool := {true, false}

When defining a function body, the `[:]` is often implicit:

and(true, true) = true   <=>   and([:] true, [:] true) = [:] true

Most programming languages uses `→` at type level and `=` at value level:

and(bool, bool) → bool	and(bool, bool) = bool	and : bool, bool → bool
and([:] true, [:] true) → [:] true	and([:] true, [:] true) = [:] true	and(true, true) = true

In path semantics, it does not matter which notation one uses.