Multiparadigmality
MeTTa programs organically combine elements of functional, logical and probabilistic programming providing a synergetic framework for representing declarative and procedural knowledge.
Atomspace
Each MeTTa program is represented as a subgraph of an Atomspace metagraph, and operates centrally by querying and rewriting portions of Atomspaces.

Self-modification
MeTTa handles highly abstract constructs like run-time self-modifying code simply and naturally. Programs are fully self-reflective – we can read/modify the code inside the programs.

Gradual dependent types
Type system is one of the most important features in terms of application of MeTTa language. Built-in mathematical reasoning by supporting a state-of-the-art type system.

Neural-symbolic integration
MeTTa is capable to support neural-symbolic reasoning and handling uncertainties, using probabilistic logical reasoning.

Inference engine
MeTTa is essentially nondeterministic that turns its interpreter into an inference engine. The language supports implementing different inference systems, from probabilistic programming to fuzzy logic.

Tool for AGI
With its open architecture MeTTa embraces very different AI strategies and is intended both for humans to script portions of AGI cognitive processes, and for the programming activity of AGI-related learning and reasoning algorithms themselves.

DSL for AI DSLs
MeTTa forms the ‘universal translator’ that enables a wide range of AI systems to dynamically collaborate by the creation of compatible Domain Specific Languages within one framework.

OpenCog Hyperon
MeTTa is the language of the cognitive architecture of OpenCog Hyperon. It functions as the firmware of the wildly variating components that Hyperon is made of and it is the glue that holds everything together.

Introduction to evaluation in MeTTa
MeTTa from Ground Up: Patterns of Knowledge
Basics of Types and Metatypes
Stdlib overview
Basics of Functional Programming in MeTTa
Using MeTTa from Python
Metta-Motto

Introduction to evaluation in MeTTa

Table of Contents

Main concepts

Atoms and knowledge graphs

MeTTa (Meta Type Talk) is a multi-paradigm language for declarative and functional computations over knowledge (meta)graphs.
Every MeTTa program lives inside of a particular
Atomspace
(or just
Space
if we don't insist on a particular internal representation). Atomspace is a part of the
OpenCog (Hyperon)
software ecosystem and it is essentially a knowledge database with the associated query engine to fetch and manipulate that knowledge. MeTTa programs can contain both factual knowledge and rules or functional code to perform reasoning on knowledge including programs themselves making the language fully self-reflective. One can draw an analogy with Prolog, which programs can also be considered as a knowledge base content, but with less introspective and more restrictive representation.
In an Atomspace, an
Atom
is a fundamental building block of all the data. In the context of graph representation, an Atom can be either a node or a link. In an Atomspace as metagraph, links can connect not only nodes, but other links, that is, they connect atoms, and they can connect any number of atoms (in contrast to ordinary graphs). In MeTTa as a programming language, atoms play the role of terms.
In the context of AI, Atoms can represent anything from objects, to concepts, to processes, functions or relationships. This enables the creation of rich, complex models of knowledge and reasoning.
Atom kinds and types

There are 4 kinds of Atoms in MeTTa:
Symbol , which represents some idea or concept. Two symbols having the same name are considered equal and representing the same concept. Names of symbols can be arbitrary strings. Nearly anything can be a symbol, e.g., A f known? replace-me ≵ , A f known? replace-me ≵ , A f known? replace-me ≵ , A f known? replace-me ≵ , A f known? replace-me ≵ , etc.
Expression , which can encapsulate other atoms including other expressions. Basic MeTTa syntax is Scheme-like, e.g. (f A) (implies (human Socrates) (mortal Socrates)) , (f A) (implies (human Socrates) (mortal Socrates)) , etc.
Variable , which is used to create patterns (expressions with variables). Such patterns can be matched against other atoms to assign some specific binding to their

variables. Variables are syntactically distinguished by a leading $ $x $_
$my-argument (Parent $x $y) (Implies (Human $x) (Mortal $x)) (:- (And (Implies $x
$y) (Fact $x)) $y) , e.g. $ $x $_ $my-argument (Parent $x $y) (Implies (Human $x)
(Mortal $x)) (:- (And (Implies $x $y) (Fact $x)) $y) , $ $x $_ $my-argument (Parent
$x $y) (Implies (Human $x) (Mortal $x)) (:- (And (Implies $x $y) (Fact $x)) $y) , $
$x $_ $my-argument (Parent $x $y) (Implies (Human $x) (Mortal $x)) (:- (And (Implies
$x $y) (Fact $x)) $y) , which tells the parser to convert a symbol to a variable.
Patterns could be $ $x $_ $my-argument (Parent $x $y) (Implies (Human $x) (Mortal
$x)) (:- (And (Implies $x $y) (Fact $x)) $y) , $ $x $_ $my-argument (Parent $x $y)
(Implies (Human $x) (Mortal $x)) (:- (And (Implies $x $y) (Fact $x)) $y) , $ $x $_
$my-argument (Parent $x $y) (Implies (Human $x) (Mortal $x)) (:- (And (Implies $x
$y) (Fact $x)) $y) , or any other symbolic expression with variables. Such patterns
get meaning when they are matched against expressions in the Atomspace.
Grounded , which represents sub-symbolic data in the Atomspace. It may contain any
binary object, for example operation (including deep neural networks), collection or
value. Grounded value type creators can define custom type, execution and matching
logic for the value. There are some grounded atoms in the standard library to deal
with numbers or strings, e.g. (+ 1 2) + 1 2 is an expression composed of a grounded
atom (+ 1 2) + 1 2 , which refers to an arithmetic operation, and (+ 1 2) + 1 2 and
(+ 1 2) + 1 2 , which are grounded atoms containing specific values. Adding custom
grounded atoms is a standard way for extending MeTTa and its interoperability.
Symbol
,
Variable
,
Grounded
can be considered as nodes, while
Expression
can be considered as a generalized link. This interpretation of atoms plays an
important role in MeTTa applications and Hyperon as a cognitive architecture, but is
not essential for understanding MeTTa as a programming language.
MeTTa has optional typing, which is close enough to gradual dependent types,
although with some peculiarities.
%Undefined%
is used for untyped expressions, while other types are represented as custom symbols
and expressions.
Symbol
,
Variable
,
Grounded
, and
Expression
are metatypes, which can be used to analyze MeTTa programs by themselves. They are
subtypes of
Atom
.
Special symbols

There is a small number of built-in symbols which determine how a MeTTa program will

be evaluated:
Equality symbol = defines evaluation rules for expressions and can be read as “can be evaluated as” or “can be reduced to”.
Colon symbol : is used for type declarations.
Arrow symbol -> defines type restrictions for evaluable expressions.
These atoms are of
Symbol
metatype, and do not refer to particular binary objects unlike
Grounded
atoms, but they are processed by the interpreter in a special way.
Basic evaluation

MeTTa programs

Programs in MeTTa consist of a number of atoms (mostly expressions, but individual symbols or grounded atoms can also be put there). A MeTTa script is a textual representation of the program, which is parsed atom-by-atom, and put into a program Space.
In particular, binary objects wrapped into grounded atoms are constructed from their textual representation in the course of parsing. For example,
+
and
1.05
will be turned into grounded atoms containing corresponding operation and value.
Particular grounded atoms and their textual representation is not a part of the core MeTTa language, but is defined in modules (both built-in and custom). How modules and grounded atoms are introduced is discussed in another tutorial.
If a programmer wants some atom to be evaluated immediately instead of adding it to the Space,
!
should be put before it. The result of evaluation will not be added to the Space, but will be included into the output result of the whole program.
MeTTa scripts can also have comments, starting with
;
, which will be ignored by the parser.
In the following program, the first two atoms will be added to the program space, while the next two expressions will be immediately evaluated and appear in the output.
metta
; This line will be ignored. ; This line will be ignored. Hello ; This symbol will be added to the Space Hello ; This symbol will be added to the Space (Hello World) ; This expression will also be added (Hello World) ; This expression will also be added ! (+ 1 2) ; This expression will be immediatedly evaluated ! (+ 1 2) ; This expression will be immediatedly evaluated ! (Hi there) ; as well as this one ! (Hi there) ; as well as this one
Run
If an expression starts with a grounded atom containing an operation, this operation is executed on the other elements of the tuple acting as its arguments.
(+ 1 2)
is naturally evaluated to

3
.
At the same time,
(Hi there)
is evaluated to itself, because
Hi
is not a grounded operation, but just a custom symbol. It acts similar to a data constructor in Haskell (more on this in another tutorial). Let us consider how to do computations over symbolic expressions in MeTTa.

Equalities

For a symbolic expression in MeTTa to be evaluated into something different from itself, an equality should be defined. Equality expressions work similar to function definitions in other languages. There is a number of important differences, though. Let us consider a few examples.
A nullary function simply returns its body
metta
```
( = ( h ) ( Hello world )) ( = ( h ) ( Hello world )) ! ( h ) ! ( h )
```
Run
Some functions can accept only specific values of its argument. When this argument is passed, the right-hand side of the corresponding equality is returned
metta
```
( = ( only-a A ) ( Input A is accepted )) ( = ( only-a A ) ( Input A is accepted ))
! ( only-a A ) ! ( only-a A ) ! ( only-a B ) ! ( only-a B )
```
Run
Note that
(only-a B)
is not reduced. In MeTTa, functions should not be total, and there is no hard boundary between a function and a data constructor. For example, consider this program:
metta
```
! ( respond me ) ! ( respond me ) ( = ( respond me ) ( OK, I will respond )) ( = (
respond me ) ( OK, I will respond )) ! ( respond me ) ! ( respond me )
```
Run
The first
(respond me)
will remain unchanged, while the second one will be transformed.
Functions can have variables as parameters, just like in other languages.
metta
```
( = ( duplicate $ x ) ( $ x $ x )) ( = ( duplicate $ x ) ( $ x $ x )) ! ( duplicate
A ) ! ( duplicate A ) ! ( duplicate 1.05 ) ! ( duplicate 1.05 )
```
Run
The passed arguments replace corresponding variables in the right-hand part of the equality.
Its arguments can be expressions with some structure
metta
```
( = ( swap ( Pair $ x $ y )) ( Pair $ y $ x )) ( = ( swap ( Pair $ x $ y )) ( Pair $
y $ x )) ! ( swap ( Pair A B )) ; evaluates to (Pair B A) ! ( swap ( Pair A B )) ;
evaluates to (Pair B A)
```
Run

One may notice that this feature is similar to pattern matching in functional languages:
metta
(= (Cdr (Cons $ x $ xs)) $ xs) (= (Cdr (Cons $ x $ xs)) $ xs) ! (Cdr (Cons A (Cons B Nil))) ; outputs (Cons B Nil) ! (Cdr (Cons A (Cons B Nil))) ; outputs (Cons B Nil)
Run
But it is more general, because the structure of patterns can be arbitrary. In particular, patterns can contain the same variable encountered multiple times.
metta
(= (check ($ x $ y $ x)) ($ x $ y)) (= (check ($ x $ y $ x)) ($ x $ y)) ! (check (B A B)) ; reduced to (B A) ! (check (B A B)) ; reduced to (B A) ! (check (B A A)) ; not reduced ! (check (B A A)) ; not reduced
Run
Functions can have multiple (nondeterministic) results. The following code will output both 0 1 and 0 1
metta
(= (bin) 0) (= (bin) 0) (= (bin) 1) (= (bin) 1) ! (bin) ; both 0 and 1 ! (bin) ; both 0 and 1
Run
Note that equations for functions are not mutually exclusive, and the following code will output two results (not only
catched
) in the last case
metta
(= (f special-value) catched) (= (f special-value) catched) (= (f $ x) $ x) (= (f $ x) $ x) ! (f A) ; A ! (f A) ; A ! (f special-value) ; both catched and special-value ! (f special-value) ; both catched and special-value
Run
Most importantly, variables can also be passed when calling a function, unlike imperative or functional languages. This will result in returning corresponding right-hand sides of equalities.
metta
(= (brother Mike) Tom) (= (brother Mike) Tom) (= (brother Sam) Bob) (= (brother Sam) Bob) ! (brother $ x) ; just Tom and Bob are returned ! (brother $ x) ; just Tom and Bob are returned ! ((brother $ x) is the brother of $ x) ; the binding for $x is not lost ! ((brother $ x) is the brother of $ x) ; the binding for $x is not lost
Run
All these features are implemented using one mechanism, which is discussed later.
Evaluation chaining

The result of the function is evaluated further both for symbolic and grounded operation:
metta
(= (square $ x) (* $ x $ x)) (= (square $ x) (* $ x $ x)) ! (square 3) ! (square 3)
Run
Here,
(square 3)

is first reduced to
(* 3 3)
, which, in turn, is evaluated to
9
by calling the grounded operation
*
.
In the following example,
Second
deconstructs the input list and returns
Car
for its tail, which is evaluated further
metta
(= (Car (Cons $ x $ xs)) $ x) (= (Car (Cons $ x $ xs)) $ x) (= (Second (Cons $ x $ xs)) (Car $ xs)) (= (Second (Cons $ x $ xs)) (Car $ xs)) ! (Second (Cons A (Cons B Nil))) ; outputs B ! (Second (Cons A (Cons B Nil))) ; outputs B
Run
Arguments of functions will typically be evaluated before the function is called. How this behavior can be controlled is discussed in a separate tutorial. The following examples should be pretty straightforward:
metta
! (* (+ 1 2) (- 8 3)) ; 15 ! (* (+ 1 2) (- 8 3)) ; 15 (= (square $ x) (* $ x $ x)) (= (square $ x) (* $ x $ x)) ! (square (+ 2 3)) ; 25 ! (square (+ 2 3)) ; 25 (= (triple $ x) ($ x $ x $ x)) (= (triple $ x) ($ x $ x $ x)) (= (grid3x3 $ x) (triple (triple $ x))) (= (grid3x3 $ x) (triple (triple $ x))) ! (grid3x3 (square (+ 1 2))) ; ((9 9 9) (9 9 9) (9 9 9)) ! (grid3x3 (square (+ 1 2))) ; ((9 9 9) (9 9 9) (9 9 9))
Run
This behavior is not different from other, especially functional, languages.
Passing results of nondeterministic functions to other functions (both deterministic and nondeterministic) cause the outer functions to be evaluated on each result. Consider the following examples:
metta
; nondeterministic function ; nondeterministic function (= (bin) 0) (= (bin) 0) (= (bin) 1) (= (bin) 1) ; deterministic triple ; deterministic triple (= (triple $ x) ($ x $ x $ x)) (= (triple $ x) ($ x $ x $ x)) ! (triple (bin)) ; (0 0 0) and (1 1 1) ! (triple (bin)) ; (0 0 0) and (1 1 1) ; nondeterministic pair ; nondeterministic pair (= (bin2) ((bin) (bin))) (= (bin2) ((bin) (bin))) ! (bin2) ; (0 0), (0 1), (1 0), (1 1) ! (bin2) ; (0 0), (0 1), (1 0), (1 1) ; deterministic summation ; deterministic summation (= (sum ($ x $ y)) (+ $ x $ y)) (= (sum ($ x $ y)) (+ $ x $ y)) (= (sum ($ x $ y $ z)) (+ $ x (+ $ y $ z))) (= (sum ($ x $ y $ z)) (+ $ x (+ $ y $ z))) ! (sum (triple (bin))) ; 0, 3 ! (sum (triple (bin))) ; 0, 3 ! (sum (bin2)) ; 0, 1, 1, 2 ! (sum (bin2)) ; 0, 1, 1, 2 ; nondeterministic increment ; nondeterministic increment (= (inc-flip $ x) (+ 0 $ x)) (= (inc-flip $ x) (+ 0 $ x)) (= (inc-flip $ x) (+ 1 $ x)) (= (inc-flip $ x) (+ 1 $ x)) ! (inc-flip 1) ; 1, 2 ! (inc-flip 1) ; 1, 2 ! (inc-flip (bin)) ; 0, 1, 1, 2 ! (inc-flip (bin)) ; 0, 1, 1, 2
Run

(triple (bin))
produces only two results, bevause
(bin)
is evaluated first and then passed to
triple
, while
(bin2)
produces four results, because each
(bin)
in its body is evaluated independently. Deterministic
sum
simply processes each nondeterministic value of its argument, while
inc-flip
doubles the number of input values.
Recursion and control

Basic recursion

A natural way to represent repetitive computations in MeTTa is recursion like in traditional functional languages, especially for processing recursive data structures. Let us consider a very basic recursive function, which calculates the number of elements in the list.
metta
(= (length ()) 0) (= (length ()) 0) (= (length (:: $ x $ xs)) (= (length (:: $ x $ xs)) (+ 1 (length $ xs))) (+ 1 (length $ xs))) ! (length (:: A (:: B (:: C ())))) ! (length (:: A (:: B (:: C ()))))
Run
The function has two cases, which are mutually exclusive de facto, and act as a conditional control structure. The base case returns
0
for an empty list
()
. Recursion itself takes place inside the second equality, in which
length
is defined via itself on the deconstructed parameter.
Notice that we didn't define the recursive data structure (list) here, and used arbitrary atoms (
::
and
()
) as data constructors.
length
can be called on anything, e.g.
(length (hello world))
, but this expression will simply be not reduced, because there are no suitable equalities for it. You can write your own version of length for other
Cons
and
Nil
instead of

::
and
()
:
sandbox
metta
```
( = ( length ... ) 0 ) ( = ( length ... ) 0 ) ( = ( length ... ) ( = ( length ... ) ( + 1 ( length $ xs ))) ( + 1 ( length $ xs ))) ! ( length ( Cons A ( Cons B ( Cons C Nil )))) ! ( length ( Cons A ( Cons B ( Cons C Nil ))))
```
Run
Copied
Reset
If a function expects specific subset of all possible expressions as input, types for corresponding atoms should be defined. However, we focus here on the basic evaluation process itself and leave types for
another tutorial
.
Higher order functions

Higher-order functions is a powerful abstraction, which naturally appears in MeTTa. Consider the following code:
metta
```
( = ( apply-twice $ f $ x ) ( = ( apply-twice $ f $ x ) ( $ f ( $ f $ x ))) ( $ f ( $ f $ x ))) ( = ( square $ x ) ( * $ x $ x )) ( = ( square $ x ) ( * $ x $ x )) ( = ( duplicate $ x ) ( $ x $ x )) ( = ( duplicate $ x ) ( $ x $ x )) ! ( apply-twice square 2 ) ; 16 ! ( apply-twice square 2 ) ; 16 ! ( apply-twice duplicate 2 ) ; ((2 2) (2 2)) ! ( apply-twice duplicate 2 ) ; ((2 2) (2 2)) ! ( apply-twice 1 2 ) ; (1 (1 2)) ! ( apply-twice 1 2 ) ; (1 (1 2))
```
Run
apply-twice
takes a function as its first parameter and applies it twice to its second parameter. In fact, it doesn't really care if it is a function or not. It simply constructs a corresponding expression for further evaluation.
Passing functions into recursive functions is very convenient for processing various collections. Consider the following basic example
metta
```
( = ( map $ f ()) ()) ( = ( map $ f ()) ()) ( = ( map $ f ( :: $ x $ xs )) ( = ( map $ f ( :: $ x $ xs )) ( :: ( $ f $ x ) ( map $ f $ xs ))) ( :: ( $ f $ x ) ( map $ f $ xs ))) ( = ( square $ x ) ( * $ x $ x )) ( = ( square $ x ) ( * $ x $ x )) ( = ( twice $ x ) ( * $ x 2 )) ( = ( twice $ x ) ( * $ x 2 )) ! ( map square ( :: 1 ( :: 2 ( :: 3 ())))) ; (:: 1 (:: 4 (:: 9 ()))) ! ( map square ( :: 1 ( :: 2 ( :: 3 ())))) ; (:: 1 (:: 4 (:: 9 ()))) ! ( map twice ( :: 1 ( :: 2 ( :: 3 ())))) ; (:: 2 (:: 4 (:: 6 ()))) ! ( map twice ( :: 1 ( :: 2 ( :: 3 ())))) ; (:: 2 (:: 4 (:: 6 ()))) ! ( map A ( :: 1 ( :: 2 ( :: 3 ())))) ; (:: (A 1) (:: (A 2) (:: (A 3) ()))) ! ( map A ( :: 1 ( :: 2 ( :: 3 ())))) ; (:: (A 1) (:: (A 2) (:: (A 3) ())))
```
Run
map
transforms a list by applying a given function (or constructor) to each element.
There is a rich toolset of higher-order functions in functional programming. They are covered in

another tutorial
.
Conditional statements

Let us imagine that we want to implement the factorial operation. If we want to use grounded arithmetics, we will not be able to use pattern matching to deconstruct a grounded number and distinguish the base and recursive cases. We can write
(= (fact 0) 1)
, but we cannot just write
(= (fact $x) (* $x (fact (- $x 1))))
. However, we can use
if
, which works much like if-then-else construction in any other language. Consider the following code
metta
(= (factorial $ x) (= (factorial $ x) (if (> $ x 0) (if (> $ x 0) (* $ x (factorial (- $ x 1))) (* $ x (factorial (- $ x 1))) 1)) 1)) ! (factorial 5) ; 120 ! (factorial 5) ; 120
Run
(factorial $x)
will be reduced to
(* $x (factorial (- $x 1)))
if
(> $x 0)
is
True
, and to
1
otherwise.
It should be noted that
if
doesn't evaluate all its arguments, but "then" and "else" branches are evaluated only when needed.
factorial
wouldn't work otherwise, although this should be more obvious from the following code, which will not execute the infinite loop
metta
(= (loop) (loop)) ; this is an infinite loop (= (loop) (loop)) ; this is an infinite loop ! (if True Success (loop)) ; Success ! (if True Success (loop)) ; Success
Run
Application of
if
looks like as an ordinary function application, and
if
is indeed implemented in pure MeTTa as a function. How it is done is discussed in another tutorial
.
Another conditional statement in MeTTa is
case

, which pattern-matches the given atom against a number of patterns sequentially in a mutually exclusive way. A different version of the factorial operation can be implemented with it:
metta
(= (factorial $ x) (= (factorial $ x) (case $ x (case $ x ((0 1) ((0 1) ($ _ (* $ x (factorial (- $ x 1))))) ($ _ (* $ x (factorial (- $ x 1))))))))) ! (factorial 5) ; 120 ! (factorial 5) ; 120
Run
In contrast to
if
,
case
doesn't check logical conditions but performs pattern matching similar to application of a function with several equality definitions. Thus, their usage is somewhat different. For example, if one wants to zip two lists, it is convenient to distinguish two cases - when both lists are empty, and both lists are not empty. But when two lists are of different lengths, there will a situation when neither of these cases will be applicable, and the expression will not be reduced. Try to run this code:
sandbox
metta
(= (zip () ()) ()) (= (zip () ()) ()) (= (zip (:: $ x $ xs) (:: $ y $ ys)) (= (zip (:: $ x $ xs) (:: $ y $ ys)) (:: ($ x $ y) (zip $ xs $ ys))) (:: ($ x $ y) (zip $ xs $ ys))) ! (zip (:: A (:: B ())) (:: 1 (:: 2 ()))) ; (:: (A 1) (:: (B 2) ())) ! (zip (:: A (:: B ())) (:: 1 (:: 2 ()))) ; (:: (A 1) (:: (B 2) ())) ! (zip (:: A (:: B ())) (:: 1 ())) ; (:: (A 1) (zip (:: B ()) ())) ! (zip (:: A (:: B ())) (:: 1 ())) ; (:: (A 1) (zip (:: B ()) ()))
Run
Copied
Reset
The non-matchable part remains unreduced. Of course, adding two equalities for
(zip (:: $x $xs) ())
and
(zip () (:: $y $ys))
could be used (you can try to add them in the above code), and it would be a more preferable way in some cases. However, using
case
here could be more convenient:
metta
(= (zip $ list1 $ list2) (= (zip $ list1 $ list2) (case ($ list1 $ list2) (case ($ list1 $ list2) (((() ()) ()) (((() ()) ()) (((:: $ x $ xs) (:: $ y $ ys)) (:: ($ x $ y) (zip $ xs $ ys))) (((:: $ x $ xs) (:: $ y $ ys)) (:: ($ x $ y) (zip $ xs $ ys))) ($ else ERROR) ($ else ERROR))))))) ! (zip (:: A (:: B ())) (:: 1 (:: 2 ()))) ; (:: (A 1) (:: (B 2) ())) ! (zip (:: A (:: B ())) (:: 1 (:: 2 ()))) ; (:: (A 1) (:: (B 2) ())) ! (zip (:: A (:: B ())) (:: 1 ())) ; (:: (A 1) ERROR) ! (zip (:: A (:: B ())) (:: 1 ())) ; (:: (A 1) ERROR)
Run
Free variables and nondeterminism again, recursively

A piece of logic

We have already encountered
if
, which reduces to different expressions depending on whether its first argument is
True
or
False
. They are returned by such grounded operations as
>
or
==
. There are also such common logical operations as
and
,
or
,
not
in MeTTa (see the stdlib tutorial for more information). Things start to get interesting, when we pass free variables into logical expressions.
Let us consider the following program.
sandbox
metta
; Some facts as very basic equalities ; Some facts as very basic equalities (= (croaks Fritz) True) (= (croaks Fritz) True) (= (eats_flies Fritz) True) (= (eats_flies Fritz) True) (= (croaks Sam) True) (= (croaks Sam) True) (= (eats_flies Sam) False) (= (eats_flies Sam) False) ; If something croaks and eats flies, it is a frog. ; If something croaks and eats flies, it is a frog. ; Note that if either (croaks $x) or (eats_flies $x) ; Note that if either (croaks $x) or (eats_flies $x) ; is false, (frog $x) is also false. ; is false, (frog $x) is also false. (= (frog $ x) (= (frog $ x) (and (croaks $ x) (and (croaks $ x) (eats_flies $ x))) (eats_flies $ x))) ! (if (frog $ x) ($ x is Frog) ($ x is-not Frog)) ! (if (frog $ x) ($ x is Frog) ($ x is-not Frog)) ; (green $x) is true if (frog $x) is true, ; (green $x) is true if (frog $x) is true, ; otherwise it is not calculated. ; otherwise it is not calculated. (= (green $ x) (= (green $ x) (if (frog $ x) True (empty))) (if (frog $ x) True (empty))) ! (if (green $ x) ($ x is Green) ($ x is-not Green)) ! (if (green $ x) ($ x is Green) ($ x is-not Green))
Run
Copied
Reset
There are some facts about
Fritz
and
Sam
, and there is a general rule about frogs. Just asking whether
(frog $x)
is
True
, we can infer that

Fritz
is a
Frog
, while
Sam
is not a
Frog
(detailed analysis of how it works is given in another tutorial).
(green $x)
is defined in such a way that it is
True
when
(frog $x)
is
True
. However, if
(frog $x)
is
False
, it returns
(empty)
(which is evaluated to an empty set of results, which is equivalent to not defining a function on the corresponding data). Running the above code reveals that
Fritz
is green, but we cannot say whether
Sam
is green or not.
Make the replacement in the above code with the naive version of
(green $x)
.
metta
```
( = ( green $ x ) ( = ( green $ x ) ( if ( frog $ x ) True ( empty ))) ( if ( frog $ x ) True ( empty ))) ( = ( green $ x ) ( frog $ x )) ( = ( green $ x ) ( frog $ x ))
```
This will result in
(Sam is-not Green)
to appear, which shows that
(= (green $x) (frog $x))
is not the same as logical implication even with boolean return values, although it is not precisely the same as equivalence (more on this in another tutorial).
You can also try to add
(= (eats_flies Tod) True)
into the set of facts.
(green Tod)
can be evaluated only partially (particular behavior is not fixed and might be different for different versions of MeTTa).
Recursion with nondeterminism

Let us generalize generation of random binary pairs to binary lists of a given length. Examine the following program:
metta

```
; random bit ; random bit ( = ( bin ) 0 ) ( = ( bin ) 0 ) ( = ( bin ) 1 ) ( = ( bin
) 1 ) ; binary list ; binary list ( = ( gen-bin $ n ) ( = ( gen-bin $ n ) ( if ( > $
n 0 ) ( if ( > $ n 0 ) ( :: ( bin ) ( gen-bin ( - $ n 1 ))) ( :: ( bin ) ( gen-bin (
- $ n 1 ))) ())) ())) ! ( gen-bin 3 ) ! ( gen-bin 3 )
```
Run
It will generate all the binary strings of length
3
. Similarly, functions to generate all the binary trees of the given depth, or all the strings up to a certain length can be written.
Try to write a function, which will output the binary list of the same length as an input list. You don't need to calculate the length of this list and to use
if
.
sandbox
metta
```
( = ( bin ) 0 ) ( = ( bin ) 0 ) ( = ( bin ) 1 ) ( = ( bin ) 1 ) ( = ( gen-bin-list
()) ()) ( = ( gen-bin-list ()) ()) ( = ( gen-bin-list ... ) ( = ( gen-bin-list ... )
... ) ... )  ! ( gen-bin-list ( :: 1 ( :: 5 ( :: 7 ())))) ! ( gen-bin-list ( :: 1 (
:: 5 ( :: 7 ()))))
```
Run
Copied
Reset
Solving problems with recursive nondeterminism

Let us put all the pieces together and solve the subset sum problem. In this problem, a list of integers is given, and one needs to find its elements whose sum will be equal to a given target sum. Candidate solutions in this problem can be represented as binary lists. Then, the sum of taken elements can be calculated as a sum of products of elements of two lists.
metta
```
; random bit ; random bit ( = ( bin ) 0 ) ( = ( bin ) 0 ) ( = ( bin ) 1 ) ( = ( bin
) 1 ) ; binary list with the same number of elements ; binary list with the same
number of elements ( = ( gen-bin-list ()) ()) ( = ( gen-bin-list ()) ()) ( = (
gen-bin-list ( :: $ x $ xs )) ( = ( gen-bin-list ( :: $ x $ xs )) ( :: ( bin ) (
gen-bin-list $ xs )) ( :: ( bin ) ( gen-bin-list $ xs )) ) ) ; sum of products of
elements of two lists ; sum of products of elements of two lists ( = (
scalar-product () ()) 0 ) ( = ( scalar-product () ()) 0 ) ( = ( scalar-product ( ::
$ x $ xs ) ( :: $ y $ ys )) ( = ( scalar-product ( :: $ x $ xs ) ( :: $ y $ ys )) (
+ ( * $ x $ y ) ( scalar-product $ xs $ ys )) ( + ( * $ x $ y ) ( scalar-product $
xs $ ys )) ) ) ; check the candidate solution ; check the candidate solution ( = (
test-solution $ numbers $ solution $ target-sum ) ( = ( test-solution $ numbers $
solution $ target-sum ) ( if ( == ( scalar-product $ numbers $ solution ) ( if ( ==
( scalar-product $ numbers $ solution ) $ target-sum ) $ target-sum ) $ solution $
solution ( empty ) ( empty ) ) ) ) ) ; task ; task ( = ( task ) ( :: 8 ( :: 3 ( ::
10 ( :: 17 ()))))) ( = ( task ) ( :: 8 ( :: 3 ( :: 10 ( :: 17 ()))))) ! (
test-solution ( task ) ( gen-bin-list ( task )) 20 ) ! ( test-solution ( task ) (
gen-bin-list ( task )) 20 )
```
Run
This solution is not scalable, but it illustrates the general idea of how nondeterminism and recursion can be combined for problem solving. Note that passing

a variable instead of
(gen-bin-list (task))
will not work here. What is the difference with the
frog
example? The answer will be given in the next tutorial.
MeTTa from Ground Up: Patterns of Knowledge

Table of Contents

Querying space content

Introduction

As a declarative language, MeTTa was designed for expressing complex relationships between entities of various types, performing computations on these relationships, and manipulating their structures. It allows programmers to specify AI algorithms and knowledge representations in a rich and flexible way. MeTTa code can be generated and processed in run-time by MeTTa programs themselves, which adds a lot of dynamism in working with complex data structures for AI tasks.
One of the main purposes of developing MeTTa was to operate over a knowledge metagraph called AtomSpace (or just Space), designed to store all sorts of knowledge, from raw sensory/motor data to linguistic and cultural knowledge, to abstract, mathematical, scientific or programming knowledge.
AtomSpace represents knowledge in the form of Atoms, the fundamental building block of all the data. Specifically, in the context of AI, an Atom can represent anything from objects, to concepts, to processes or relationships, to reasoning rules and algorithms.
While MeTTa may look like an ordinary language in certain aspects, it is built on top of operations over the knowledge metagraph, which is essential to understand how it works.
Knowledge declaration and matching query

Let us look at a basic example of specifying relations between concepts, e.g., family relationships. While there are different ways to do this, in MeTTa, one can simply put expressions like the following into the program
metta
(Parent Tom Bob) (Parent Tom Bob)
This expression being put into the program space can be treated as the fact that Tom is Bob's parent. We start
Parent
with capital
P
to distinguish it from a function, which we would prefer to start with
p
in this case, although this naming convention is not mandatory.
One can add more such expressions to the program space. But what can we do with such expressions? The

tutorial
overviewed the evaluation process of expressions, for which equalities are specified. But is there any use of expressions without equalities?
The core operation in MeTTa is
matching
. It searches for all declared atoms corresponding to the given pattern and produces the output pattern. The process is similar to the manner in which one can search text strings with regular expressions, but it is for searching for subgraphs in a metagraph.
We can compose a query for matching using the grounded function
match
. It expects three arguments:
a grounded atom referencing a Space;
pattern atom to be matched;
output pattern typically containing variables from the input pattern.
Basic examples

Let us consider the following program
metta
(Parent Bob Ann) (Parent Bob Ann) ; This match will be successful ; This match will be successful ! (match &self (Parent Bob Ann) (Bob is Ann`s father )) ! ( match &self ( Parent Bob Ann ) ( Bob is Ann`s father)) ; The following line will return [] ; The following line will return [] ! (match &self (Parent Bob Joe) (Bob is Joe`s father )) ! ( match &self ( Parent Bob Joe ) ( Bob is Joe`s father))
Run
&self
is a reference to the current program Space. We can refer to other Atomspaces, but we will cover it later. The second argument in the first
match
expression
(Parent Bob Ann)
is an expression to be matched against atoms in the current Space, and the third argument
(Bob is Ann's father)
is the atom to be returned if matching succeeded.
The program above will return
[(Bob is Ann's father)]
and
[]
, since when the desired expression pattern wasn't found
match
returns nothing.
We can construct more interesting queries using variables. Let us consider the program
metta
(Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) ! (match &self (Parent $ x Bob) $ x) ; [Tom, Pam] ! (match &self (Parent $ x Bob) $ x) ; [Tom, Pam]
Run

The pattern
(Parent $x Bob)
, i.e. "Who are Bob's parents?", can be matched against two atoms (facts) in the Space, and corresponding bindings for
$x
will be used to produce the result of
match
. Here, we will get two matches
[Tom, Pam]
, which can be viewed as a
nondeterministic
evaluation of
match
.
Please, note that
match
doesn't search in subexpressions. The following code will return
[Ann]
only:
metta

```
( Parent Bob Ann ) ( Parent Bob Ann ) ( Parent Pam ( Parent Bob Pat )) ( Parent Pam ( Parent Bob Pat ))  ! ( match &self ( Parent Bob $ x ) $ x ) ; Ann ! ( match &self ( Parent Bob $ x ) $ x ) ; Ann
```

Run
We can make even broader queries: "Who is a parent of whom?", or "Find
$x
and
$y
such that
$x
is a parent of
$y
".
sandbox
metta

```
( Parent Tom Bob ) ( Parent Tom Bob ) ( Parent Pam Bob ) ( Parent Pam Bob ) ( Parent Tom Liz ) ( Parent Tom Liz ) ( Parent Bob Ann ) ( Parent Bob Ann ) ( Parent Bob Pat ) ( Parent Bob Pat ) ( Parent Pat Pat ) ( Parent Pat Pat )  ! ( match &self ( Parent $ x $ y ) ( $ x $ y )) ! ( match &self ( Parent $ x $ y ) ( $ x $ y ))
```

Run
Copied
Reset
The output should contain the following pairs (the order can be different due to MeTTa's nondeterminism)
[(Pat Bob), (Bob Ann), (Bob Pat), (Tom Bob), (Tom Liz), (Pat Pat)]
. Can you add the query in the above program to retrieve only parents and children with same names?
Functions and unification

Function evaluation and matching

As discussed in the
tutorial
, evaluable expressions can contain variables, and they are pattern-matched against left-hand side of equalities. In fact, evaluation of expressions can be understood as recursively constructing queries for equalities. Consider this code as an example
metta
```
( = ( only-a A ) ( Input A is accepted )) ( = ( only-a A ) ( Input A is accepted ))
! ( only-a A ) ! ( only-a A ) ! ( only-a B ) ! ( only-a B ) ! ( only-a $ x ) ! (
only-a $ x )
```
Run
Evaluation of
(only-a A)
can be thought of as execution of query
(match &self (= (only-a A) $result) $result)
.
$result
will be bound with the right-hand side of the function case (body), if the left-hand side matches with the expression under evaluation. Does it work for
(only-a B)
and
(only-a $x)
?
Let us check that the following program produces the same result:
metta
```
( = ( only-a A ) ( Input A is accepted )) ( = ( only-a A ) ( Input A is accepted ))
! ( match &self ( = ( only-a A ) $ result ) $ result ) ! ( match &self ( = ( only-a
A ) $ result ) $ result ) ! ( match &self ( = ( only-a B ) $ result ) $ result ) ! (
match &self ( = ( only-a B ) $ result ) $ result ) ! ( match &self ( = ( only-a $ x
) $ result ) $ result ) ! ( match &self ( = ( only-a $ x ) $ result ) $ result )
```
Run
There is one difference.
match
produces the empty result in the second case, while the interpreter keeps this expression unreduced. The interpreter is performing some additional processing on top of such equality queries.
While allowing the MeTTa interpreter to construct equality queries automatically for evaluating expressions like
(only-a A)
is very convenient for functional programming, using
match
directly allows for more compact knowledge representation and efficient queries glued together in a custom way.
It should also be noted that obtaining multiple results in queries to knowledge bases is very typical, and since the semantics of evaluating expressions in MeTTa is natively related to such queries, all evaluations in MeTTa are secretly or explicitly nondeterministic.
Let us analyze the following program:
metta
```
( Parent Tom Bob ) ( Parent Tom Bob ) ( Parent Pam Bob ) ( Parent Pam Bob ) ( Parent
```

Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) (= (get-parent-entries $ x $ y) (= (get-parent-entries $ x $ y) (match &self (Parent $ x $ y) (Parent $ x $ y))) (match &self (Parent $ x $ y) (Parent $ x $ y))) (= (get-parents $ x) (= (get-parents $ x) (match &self (Parent $ y $ x) $ y)) (match &self (Parent $ y $ x) $ y)) ! (get-parent-entries Tom $ _) ! (get-parent-entries Tom $ _) ! (get-parents Bob) ! (get-parents Bob)
Run
We can call match (get-parent-entries Tom $_) (match &self (Parent Tom $y) (Parent Tom $y)) from an ordinary function, and we can still pass variable arguments to it, so match (get-parent-entries Tom $_) (match &self (Parent Tom $y) (Parent Tom $y)) is equivalent to match (get-parent-entries Tom $_) (match &self (Parent Tom $y) (Parent Tom $y)) .
The result [(Parent Tom Liz), (Parent Tom Bob)] is not reduced further. It is convenient, when we want to represent pieces of knowledge and process them.
(get-parents Bob) [Tom, Pam] match returns (get-parents Bob) [Tom, Pam] match .
Executing (get-parents Bob) [Tom, Pam] match from functions allows creating convenient functional abstractions while still working with declarative knowledge.
For example, how would you write a function, which returns grandparents of a given person?
sandbox
metta
(Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) (Parent Bob Pat) (Parent Bob Pat) (Parent Pat Jim) (Parent Pat Jim) (= (get-parents $ x) (= (get-parents $ x) (match &self (Parent $ y $ x) $ y)) (match &self (Parent $ y $ x) $ y)) (= (get-grand-parents $ x) (= (get-grand-parents $ x) (...)) (...)) ! (get-grand-parents Pat) ! (get-grand-parents Pat)
Run
Copied
Reset
From facts to rules

One may notice that equality queries for functions suppose that there are free variables not only in the query, but also in the Atomspace entries. These entries can be not only function definitions, but other arbitrary expressions, which can be used to represent general knowledge or rules. For example, one can write
metta
(Parent Tom Bob) (Parent Tom Bob) (Parent Bob Ann) (Parent Bob Ann) (Implies (Parent $ x $ y) (Child $ y $ x)) (Implies (Parent $ x $ y) (Child $ y $ x)) (= (deduce $ B) (= (deduce $ B) (match &self (Implies $ A $ B) (match &self (Implies $ A $ B) (match &self $ A $ B)) (match &self $ A $ B)))) (= (conclude $ A) (= (conclude $ A) (match &self (Implies $ A $ B) (match &self (Implies $ A $ B) (match &self $ A $ B)) (match &self $ A $ B)))) ! (deduce (Child $ x Tom)) ; [(Child Bob Tom)] ! (deduce (Child $ x Tom)) ; [(Child Bob Tom)] ! (conclude (Parent Bob $ y)) ; [(Child Ann Bob)] ! (conclude (Parent Bob $ y)) ; [(Child Ann Bob)]
Run
If
Child
and

Parent
were predicates returning
True
or
False
(as in the
frog example
), we could somehow use
=
instead of
Implies
. But here we don't evaluate the premise to
True
or
False
, but check that it is in the knowledge base. It makes inference better controllable. We can easily go from premises to conclusions with
conclude
, or to verify conclusions by searching for suitable premises with
deduce
.
We will discuss different ways of introducing reasoning in MeTTa in more detail later. What we want to focus on now is that in both cases a query with variables is constructed, say,
(Implies (Parent Bob $y) $B)
and it should be matched against some entry in the knowledge base with variables as well, namely,
(Implies (Parent $x $y) (Child $y $x))
in our example. This operation is called unification, and it is available in MeTTa in addition to
match
.
Unification

Function
unify
accepts two patterns to be unified (matched together in such the way that shared variables in them get most general non-contradictory substitutions). The function is evaluated to its third argument if unification is successful and to the fourth argument otherwise. The following program shows the basic example.
metta
! (unify (parent $ x Bob) ; the first pattern ! (unify (parent $ x Bob) ; the first pattern (parent Tom $ y) ; the second pattern (parent Tom $ y) ; the second pattern ($ x $ y) ; the output for successful unification ($ x $ y) ; the output for successful unification Fail) ; fallback Fail) ; fallback
Run
Here, we unify two expressions
(parent $x Bob)
and
(parent Tom $y)

, and return a tuple
($x $y)
if unification succeeded. The
Fail
atom will be returned if there are no matches. Note that
(unify (A $x) ($x B) Yes No)
will be reduced to
No
, because
$x
should have the same binding in both patterns (and it cannot be
A
and
B
simultaneously).
One of the first two arguments can be a reference to a Space as well. In this case, it will work like
match
but with an alternative option in the case of failed matching:
metta
(Parent Tom Bob) (Parent Tom Bob) (Parent Bob Ann) (Parent Bob Ann) ! (unify &self (Parent $ x Bob) $ x Fail) ; [Tom] ! (unify &self (Parent $ x Bob) $ x Fail) ; [Tom]
Run
Here, we pass a reference to the current Space as the first argument, so the second expression
(parent $x Bob)
is matched against the whole set of declared knowledge.
Chained unification

Let us analyze how
(conclude (Parent Bob $y))
from the above example is evaluated.
At first, (match &self (= (Parent Bob $y) $result) $result) Parent (Parent Bob $y) is executed to evaluate the subexpression. But this query returns no result, because equalities for (match &self (= (Parent Bob $y) $result) $result) Parent (Parent Bob $y) are not defined. Thus, (match &self (= (Parent Bob $y) $result) $result) Parent (Parent Bob $y) remains unreduced.
Thus, the equality query for the whole expression (match &self (= (conclude (Parent Bob $y)) $result) $result) is executed. The following two expressions (one is the query and another one is from Space) are unifiable:
metta
(= (conclude (Parent Bob $ y)) (= (conclude (Parent Bob $ y)) $ result) $ result) (= (conclude $ A) (= (conclude $ A) (match &self (Implies $ A $ B) (match &self (Implies $ A $ B) (match &self $ A $ B))) (match &self $ A $ B)))
$A
will be bound to
(Parent Bob $y)
, and

$result
will be
metta
(match &self (Implies (Parent Bob $ y) $ B) (match &self (Implies (Parent Bob $ y) $ B) (match &self (Parent Bob $ y) $ B)) (match &self (Parent Bob $ y) $ B))
match (Implies (Parent Bob $y) $B) is executed directly as a grounded function (otherwise another equality query would be constructed) with match (Implies (Parent Bob $y) $B) as a query. It unifies with the following entry in the Space:
metta
(Implies (Parent Bob $ y) $ B) (Implies (Parent Bob $ y) $ B) (Implies (Parent $ x $ y) (Child $ y $ x)) (Implies (Parent $ x $ y) (Child $ y $ x))
One may notice that there could be some collisions of variable names, and the interpreter should deal with this. In overall,
$x
gets bound to
Bob
, and
$B
gets bound to
(Child $y Bob)
. Since the output of this
match
is
(match &self (Parent Bob $y) $B)
, the expression for further evaluation becomes
(match &self (Parent Bob $y) (Child $y Bob))
.
(Parent Bob $y) (Parent Bob Ann) (Child Ann Bob) unifies with (Parent Bob $y) (Parent Bob Ann) (Child Ann Bob) yielding (Parent Bob $y) (Parent Bob Ann) (Child Ann Bob)
Query (= (Child Ann Bob) $result) (Child Ann Bob) finds no matches, so (= (Child Ann Bob) $result) (Child Ann Bob) is the final result.
The overall chain of transformations in the course of interpretation can be viewed as:
metta
1. (conclude (Parent Bob $ y)) 1. (conclude (Parent Bob $ y)) 2. (match &self (Implies (Parent Bob $ y) $ B) 2. (match &self (Implies (Parent Bob $ y) $ B) (match &self (Parent Bob $ y) $ B)) (match &self (Parent Bob $ y) $ B)) 3. (match &self (Parent Bob $ y) (Child $ y Bob)) 3. (match &self (Parent Bob $ y) (Child $ y Bob)) 4. (Child Ann Bob) 4. (Child Ann Bob)
These are not all the steps done by the interpreter, but they give the overall picture of what is really going on under the hood.
Nested queries and recursive graph traversal

Composite queries

We've already seen queries for
conclude
and

deduce
, which result is another query. At the same time, chaining of queries can be done in a more functional style with equalities as it could be done for
metta
(= (get-grand-parents $ x) (= (get-grand-parents $ x) ((get-parents (get-parents $ x)))) ((get-parents (get-parents $ x))))
Keeping knowledge declarative can be useful for implementing reasoning over it. Imagine that we add more info on people like
(Female Pam)
or
(Male Tom)
into the knowledge base, and want to define more relations such as
sister
. One can turn facts into equalities like
(= (Female Pam) True)
and use functional logic (as in the
frog example
), but let us keep simple facts for now.
One can add more functions like
get-parents
. A function for
female
would be more convenient to represent as a filter, e.g.
(= (female $x) (match &self (Female $x) $x))
, so it will be composable, e.g.
(= (get-mother $x $y) (female (get-parents $x $y)))
.
One can do this by a composite query instead.
sandbox
metta
(Female Pam) (Female Pam) (Male Tom) (Male Tom) (Male Bob) (Male Bob) (Female Liz) (Female Liz) (Female Pat) (Female Pat) (Female Ann) (Female Ann) (Male Jim) (Male Jim) (Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) (Parent Bob Pat) (Parent Bob Pat) (Parent Pat Jim) (Parent Pat Jim) (= (get-sister $ x) (= (get-sister $ x) (match &self (match &self (, (Parent $ y $ x) (, (Parent $ y $ x) (Parent $ y $ z) (Parent $ y $ z) (Female $ z)) (Female $ z)) $ z $ z)))) ! (get-sister Bob) ! (get-sister Bob)
Run
Copied
Reset
Composite queries contain a few patterns (united by
,
into one expression), which should be satisfied simultaneously. Such queries can be efficient if the Atomspace query engine efficiently processes joints. This can be important for large knowledge bases. Otherwise, it is necessary to be careful about the order of nested queries of filters. For example, having
(Female $z)
with free variable

$z
as the innermost functional call or
(match &self (Female $z) ...)
as the outermost query in a nested sequence of queries will be highly inefficient, because it will first extract all the females from the knowledge base, and only then will narrow down the set of the results.
Notice that the above program is imprecise. How can the mistake be fixed (check the sister of
Liz
- one option would be to introduce
(different $x $y)
as a filter)? You can try implementing other relations like
uncle
in the above program or rewrite it in a functional way. Typically, we would like to represent such concepts using other derived concepts rather than monolithic composite queries (e.g. "Uncle is a brother of a parent" rather than "Uncle is a male child of a parent of a parent, but not the parent").
Recursion for graph traversal

Let us define the predecessor relation:

```
For any `$x` and `$z`: `$x` is a predecessor of `$z`
    if there is `$y` such that
       `$y` is a parent of `$z` and
       `$x` is a predecessor of `$y`
```

Recursion is a convenient way to represent such relations.
metta
(Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) (Parent Bob Pat) (Parent Bob Pat) (Parent Pat Jim) (Parent Pat Jim) (Parent Jim Lil) (Parent Jim Lil) (= (parent $ x $ y) (match &self (Parent $ x $ y) $ x)) (= (parent $ x $ y) (match &self (Parent $ x $ y) $ x)) (= (predecessor $ x $ z) (parent $ x $ z)) (= (predecessor $ x $ z) (parent $ x $ z)) (= (predecessor $ x $ z) (predecessor $ x (parent $ y $ z))) (= (predecessor $ x $ z) (predecessor $ x (parent $ y $ z))) ; Who are predecessors of Lil ; Who are predecessors of Lil ! (predecessor $ x Lil) ! (predecessor $ x Lil)
Run
Basics of Types and Metatypes

Table of Contents

Concrete types

Types of symbols

Atoms in MeTTa are typed. Types of atoms are also represented as atoms (typically, symbolic atoms and expressions). Expressions of the form

(: <atom> <type>)
are used to assign types. For example, to designate that the symbol atom
a
has a custom type
A
one needs to add the expression
(: a A)
to the space (program).
Note that since
A
here is a symbol atom, it can also have a type, e.g.,
(: A Type)
. The symbol atom
Type
is conventionally used in MeTTa to denote the type of type atoms. However, it is not assigned automatically. That is, declaration
(: a A)
doesn't force
A
to be of type
Type
.
When an atom has no assigned type, it has
%Undefined%
type. The value of
%Undefined%
type can be type-checked with any type required.
One can check the type of an atom with
get-type
function from stdlib.
metta
```
( : a A ) ( : a A ) ( : b B ) ( : b B ) ( : A Type ) ( : A Type )  ! ( get-type a )
; A ! ( get-type a ) ; A ! ( get-type b ) ; B ! ( get-type b ) ; B ! ( get-type c )
; %Undefined% ! ( get-type c ) ; %Undefined% ! ( get-type A ) ; Type ! ( get-type A
) ; Type ! ( get-type B ) ; %Undefined% ! ( get-type B ) ; %Undefined%
```
Run
Here, we declared types
A
and
B
for
a
and
b
correspondingly, and type
Type
for
A
.
get-type

returns the declared types or
%Undefined%
if no type information is provided for the symbol.
Types of expressions

Consider the following program.
metta
(: a A) (: a A) (: b B) (: b B) ! (get-type (a b)) ; (A B) ! (get-type (a b)) ; (A B)
Run
The type of expression
(a b)
will be
(A B)
. The type of a tuple is a tuple of types of its elements. However, what if we want to apply a function to an argument? Usually, we want to check if the function argument is of appropriate type. Also, while function applications themselves are expressions, they are transformed in the course of evaluation, and the result has its own type. Basically, we want to be able to transform (or reduce) types of expressions before or without transforming expressions themselves.
Arrow
->
is a built-in symbol of the type system in MeTTa, which is used to create a function type, for example
(: foo (-> A B))
. This type signature says that
foo
can accept an argument of type
A
and its result will be of type
B
:
metta
(: a A) (: a A) (: foo (-> A B)) (: foo (-> A B)) ! (get-type (foo a)) ; B ! (get-type (foo a)) ; B
Run
Let us note that
We didn't provide a body for foo (foo a) B foo a foo , so foo (foo a) B foo a foo is not reduced at all, and its type foo (foo a) B foo a foo is derived purely from the types of foo (foo a) B foo a foo and foo (foo a) B foo a foo . It doesn't matter whether foo (foo a) B foo a foo is a real function or a data constructor.
Equality queries themselves don't care about the position of the function symbol in the tuple, and the following code is perfectly correct
metta
(= ($ 1 infix-f $ 2) ($ 2 $ 1)) (= ($ 1 infix-f $ 2) ($ 2 $ 1)) ! (match &self (= (1 infix-f 2) $ r) $ r) ! (match &self (= (1 infix-f 2) $ r) $ r)
Run
However, reduction of the type of a tuple is performed if its
first

element has an arrow (function) type. For convenience and by convention, the first element in a tuple is treated specially for function application.

Type-checking

Types can protect against incorrectly constructed expressions including misuse of a function, when we want it to accept arguments of a certain type.

metta

```
; This function accepts an atom of type A and returns an atom of type B ; This function accepts an atom of type A and returns an atom of type B ( : foo ( -> A B )) ( : foo ( -> A B )) ( : a A ) ( : a A ) ( : b B ) ( : b B )  ! ( foo a ) ; no error ! ( foo a ) ; no error ! ( get-type ( foo b )) ; no result ! ( get-type ( foo b )) ; no result ! ( b foo ) ; notice: no error ! ( b foo ) ; notice: no error ! ( get-type ( b foo )) ; (B (-> A B)) ! ( get-type ( b foo )) ; (B (-> A B)) ! ( foo b ) ; type error ! ( foo b ) ; type error
```

Run

We didn't define an equality for
foo
, so
(foo a)
reduces to itself. However, an attempt to evaluate
(foo b)
results in the error expression. When we try to get the type of this expression with
(get-type (foo b))
, the result is empty meaning that this expression has no valid type.

Notice that evaluation of
(b foo)
doesn't produce an error. The arrow type of
foo
in the second position of the tuple doesn't cause transformation of its type.
Indeed,
(get-type (b foo))
produces
(B (-> A B))
.

Gradual typing

Let us consider what types will expressions have, when some of their elements are
%Undefined%
. Run the following program to check the currently implemented behavior

metta

```
( : foo ( -> A B )) ( : foo ( -> A B )) ( : a A ) ( : a A ) ! ( get-type ( foo c )) ! ( get-type ( foo c )) ! ( get-type ( g a )) ! ( get-type ( g a ))
```

Run

Note that
g
and
c
are of
%Undefined%
type, while

foo
and
a
are typed. The result can be different depending on which type is not defined, of the function or its argument.
Multiple arguments

Functions can have more than one argument. In their type signature, types of their parameters are listed first, and the return type is put at the end much like for functions with one argument.
The wrong order of arguments with different types as well as the wrong number of arguments will render the type of the whole expression to be empty (invalid).
sandbox
metta
; This function takes two atoms of type A and B and returns an atom of type C ; This function takes two atoms of type A and B and returns an atom of type C (: foo2 (-> A B C)) (: foo2 (-> A B C)) (: a A) (: a A) (: b B) (: b B) ! (get-type (foo2 a b)) ; C ! (get-type (foo2 a b)) ; C ! (get-type (foo2 b a)) ; empty ! (get-type (foo2 b a)) ; empty ! (get-type (foo2 a)) ; empty ! (get-type (foo2 a)) ; empty ! (foo2 a c) ; no error ! (foo2 a c) ; no error ! (foo2 b a) ; type error (the interpreter stops on error) ! (foo2 b a) ; type error (the interpreter stops on error) ! (foo2 c) ; would also be type error ! (foo2 c) ; would also be type error
Run
Copied
Reset
Here, the atom
c
is of
%Undefined%
type and it can be matched against an atom of any other type. Thus,
(foo2 a c)
will not produce an error. However,
(foo2 c)
will not work because of wrong arity.
Also notice that it is not necessary to define an instance of type
C
.
foo2
by itself acts as a constructor for this type.
What will be the type of a function with zero arguments? Its type expression will have only the return type after
->
, e.g.
metta
(: a A) (: a A) (: const-a (-> A)) (: const-a (-> A)) (= (const-a) a) (= (const-a) a)
Nested expressions

Types of nested expressions are inferred from innermost expressions outside. You can

try nesting typed expressions in the sandbox below and see what goes wrong.
sandbox
metta
```
( : foo ( -> A B )) ( : foo ( -> A B )) ( : bar ( -> B B A )) ( : bar ( -> B B A ))
( : a A ) ( : a A ) ! ( get-type ( bar ( foo a ) ( foo a ))) ! ( get-type ( bar (
foo a ) ( foo a ))) ! ( get-type ( foo ( bar ( foo a ) ( foo a )))) ! ( get-type (
foo ( bar ( foo a ) ( foo a ))))
```
Run
Copied
Reset
Note that type signatures can be nested expressions by themselves:
metta
```
( : foo-pair ( -> ( A B ) C )) ( : foo-pair ( -> ( A B ) C )) ( : a A ) ( : a A ) (
: b B ) ( : b B )  ! ( get-type ( foo-pair a b )) ; empty ! ( get-type ( foo-pair a
b )) ; empty ! ( get-type ( foo-pair ( a b ))) ; C ! ( get-type ( foo-pair ( a b )))
; C
```
Run
As was mentioned above, an arrow type of the atom, which is not the first in the tuple, will not cause type reduction. Thus, one may apply a function to another function (or a data constructor):
metta
```
( : foo ( -> ( -> A B ) C )) ( : foo ( -> ( -> A B ) C )) ( : bar ( -> A B )) ( :
bar ( -> A B )) ( : a A ) ( : a A )  ! ( get-type ( foo bar )) ; C ! ( get-type (
foo bar )) ; C ! ( get-type ( foo ( bar a ))) ; empty ! ( get-type ( foo ( bar a )))
; empty
```
Run
Here, the type of
bar
matches the type of the first parameter of
foo
. Thus,
(foo bar)
is a well-typed expression, which overall type corresponds to the return type of
foo
, namely,
C
.
(foo (bar a))
, in turn, is badly typed, because the type of
(bar a)
is reduced to
B
, which does not correspond to
(-> A B)
expected by
foo
.
Similarly, the return type of a function can be an arbitrary expression including arrow types. Try to construct a well-typed expression involving all the following symbols

sandbox
metta
(: foo (-> C (-> A B))) (: foo (-> C (-> A B))) (: bar (-> B A)) (: bar (-> B A)) (: a A) (: a A) (: c C) (: c C) ! (get-type (...)) ! (get-type (...))
Run
Copied
Reset
We intentionally don't provide function bodies here to underline that typing imposes purely structural restrictions on expressions, which don't require understanding the semantics of functions. In the example above,
foo
accepts an atom of type
C
. Thus,
(foo c)
is well-typed, and its reduced type is
(-> A B)
. This is an arrow type meaning that we can put this expression at the first position of a tuple (function application), and it will expect an atom of type
A
. Thus,
((foo c) a)
should be well-typed, and its reduced type will be
B
. Thus, we can apply
bar
to it. Will
(bar ((foo c) a))
be indeed well-typed?
Grounded atoms

Grounded atoms are also typed. One can check their types with
get-type
as well:
metta
! (get-type 1) ; Number ! (get-type 1) ; Number ! (get-type 1.1) ; Number ! (get-type 1.1) ; Number ! (get-type +) ; (-> Number Number Number) ! (get-type +) ; (-> Number Number Number) ! (get-type (+ 1 2.1)) ; Number ! (get-type (+ 1 2.1)) ; Number
Run
As the example shows,
1
and
1.1
both are of
Number
type, although their data-level representation can be different.
+
accepts two arguments of

Number
type and returns the result of the same type. Thus,
Number
is repeated three times in its type signature.
Let us note once again that the argument of
get-type
is not evaluated, and
get-type
returns an inferred type of expression. In particular, when we try to apply
+
to the argument of a wrong type, the result is the error expression (which by itself is well-typed), but
get-type
returns the empty result instead of returning the type of the error message:
metta
(: a A) (: a A) ! (get-type (+ 1 a)) ; empty ! (get-type (+ 1 a)) ; empty ! (get-type (+ 1 b)) ; Number ! (get-type (+ 1 b)) ; Number ! (+ 1 b) ; no error, not reduced ! (+ 1 b) ; no error, not reduced ! (+ 1 a) ; type error ! (+ 1 a) ; type error
Run
In this program, we also tried to see the type of application of the grounded function to the argument of
%Undefined%
type. Such the expression type-checks. However, it is not reduced in the course of evaluation. Thus, grounded functions work as partial functions or expression constructors in such cases. MeTTa is a symbolic language, and the possibility to construct expressions for further analysis is one of its main features. Ultimately, grounded functions should not differ from symbolically defined functions in this regard.
Recursive and parametric types

Recursive data types

All types allow constructing recursive expressions, when there is at least one function accepting and returning values of this type. This is true for arithmetic expressions or compositions of operations over strings. Say, any expression like
(+ (- 3 1) (* 2 (+ 3 4)))
will be of
Number
type. We expect that the result of evaluation of such expressions will have the same type as the reduced type of the expression itself.
However, in some cases, we don't even want such expressions to be reduced, but want to consider them as instances of the reduced type. Consider the simple example of Peano numbers:
metta
(: Z Nat) ; Z is "zero" (: Z Nat) ; Z is "zero" (: S (-> Nat Nat)) ; S "constructs" the next number (: S (-> Nat Nat)) ; S "constructs" the next number ! (S Z) ; this is "one" ! (S Z) ; this is "one" ! (S (S Z)) ; this is "two" ! (S (S Z)) ; this is "two" ! (get-type (S (S (S Z)))) ; Nat ! (get-type (S (S (S Z)))) ; Nat ! (get-type (S S)) ; not Nat ! (get-type (S S)) ; not Nat

Run
We didn't define the type of
Nat
itself. One may prefer to add
(: Nat Type)
for clarity.
In the code above,
S
does nothing. It could be a grounded function, which adds
1
to the given number in some binary representation. Instead,
(S some-nat)
is not reduced and serves itself to represent the next natural number. It doesn't actually important that
S
is not a function, and
(S some-nat)
is not calculated. In fact, it could be. What really matters is that instances of
Nat
can be deconstructed and pattern-matched.
The following code shows, how
Nat
as a recursive data type is processed by pattern matching.
metta
```
( : Z Nat ) ( : Z Nat ) ( : S ( -> Nat Nat )) ( : S ( -> Nat Nat )) ( : Greater ( -> Nat Nat Bool )) ( : Greater ( -> Nat Nat Bool )) ( = ( Greater ( S $ x ) Z ) ( = ( Greater ( S $ x ) Z ) True ) True ) ( = ( Greater Z $ x ) ( = ( Greater Z $ x ) False ) False ) ( = ( Greater ( S $ x ) ( S $ y )) ( = ( Greater ( S $ x ) ( S $ y )) ( Greater $ x $ y )) ( Greater $ x $ y )) ! ( Greater ( S Z ) ( S Z )) ; False ! ( Greater ( S Z ) ( S Z )) ; False ! ( Greater ( S ( S Z )) ( S Z )) ; True ! ( Greater ( S ( S Z )) ( S Z )) ; True
```
Run
While this implementation is inefficient for computations, it is more suitable for reasoning.
More practical use of recursive data structures is in the form of containers to store data. We already constructed them in the previous tutorials, but without types. Let us add typing information and define the type of list of numbers:
metta
```
( : NilNum ListNum ) ( : NilNum ListNum ) ( : ConsNum ( -> Number ListNum ListNum )) ( : ConsNum ( -> Number ListNum ListNum )) ! ( get-type ( ConsNum 1 ( ConsNum 2 ( ConsNum 3 NilNum )))) ; ListNum ! ( get-type ( ConsNum 1 ( ConsNum 2 ( ConsNum 3 NilNum )))) ; ListNum ! ( ConsNum 1 ( ConsNum "S" NilNum )) ; BadType ! ( ConsNum 1 ( ConsNum "S" NilNum )) ; BadType
```
Run
The type reduction for such expressions is rather straightforward: the type of
(ConsNum 3 NilNum)
is reduced to
ListNum
, since
ConsNum

is of
(-> Number ListNum ListNum)
type and its arguments are of
Number
and
ListNum
types. Consequently,
(ConsNum 2 (...))
is reduced to
ListNum
again for the same reason, and so on. For the second case,
(ConsNum "S" NilNum)
is badly typed.
Such expressions can be recursively processed as was done in the
tutorial
. Adding type information makes the purpose of the corresponding functions clearer and allows detecting mistakes.
Parametric types

Type expressions can contain variables. Type-checking for such types is implemented and can be understood via pattern-matching. Let us consider some basic examples.
Stdlib contains a comparison operator
==
. The following code
metta
! (get-type ==) ! (get-type ==) ! (== 1 "S") ! (== 1 "S")
Run
will reveal that
(== 1 +)
is badly typed, and the reason is that
==
has the type
(-> $t $t Bool)
. This means that the arguments can be of an arbitrary but same type. Type-checking and reduction can be understood here as an attempt to unify
(-> $t $t Bool)
with
(-> Number String $result)
.
It deserves noting that the output type can also be variable, e.g.
metta
(: apply (-> (-> $ tx $ ty) $ tx $ ty)) (: apply (-> (-> $ tx $ ty) $ tx $ ty)) (= (apply $ f $ x) ($ f $ x)) (= (apply $ f $ x) ($ f $ x)) ! (apply not False) ; True ! (apply not False) ; True ! (get-type (apply not False)) ; Bool ! (get-type (apply not False)) ; Bool ! (unify (-> (-> $ tx $ ty) $ tx $ ty) ! (unify (-> (-> $ tx $ ty) $ tx $ ty) (-> (-> Bool Bool) Bool $ result) (-> (-> Bool Bool) Bool $ result) $ result $ result BadType) ; Bool BadType) ; Bool ! (apply not 1) ; BadType ! (apply not 1) ; BadType
Run
not

has
(-> Bool Bool)
type and
False
is of
Bool
type. Thus, arguments of
(apply not False)
suppose that the function type signature should be unified with
(-> (-> Bool Bool) Bool $result)
. This results in binding both
$tx
and
$ty
to
Bool
, and the output type (
$ty
) also becomes
Bool
.
In the
tutorial
, we defined
apply-twice
, which takes the function as an argument and applies it two time to the second argument. But what if the output type of the function is different from its input type? Can it be applied to the result of its own application? Try to specify the type of
apply-twice
to catch the error in the last expression:
sandbox
metta
(: apply-twice (-> ? ? ?)) (: apply-twice (-> ? ? ?)) (= (apply-twice $ f $ x) (= (apply-twice $ f $ x) ($ f ($ f $ x))) ($ f ($ f $ x))) (: greater-than-0 (-> Number Bool)) (: greater-than-0 (-> Number Bool)) (= (greater-than-0 $ x) (> $ x 0)) (= (greater-than-0 $ x) (> $ x 0)) ! (get-type (apply-twice not True)) ; should be [Bool] ! (get-type (apply-twice not True)) ; should be [Bool] ! (get-type (apply-twice greater-than-0 1)) ; should be [] ! (get-type (apply-twice greater-than-0 1)) ; should be []
Run
Copied
Reset
Besides defining higher-order functions, parametric types are useful for recursive data structures. One of the most common examples is
List
. How can we define it as a container of elements of an arbitrary but same type? We can parameterize the type
List
itself with the type of its elements:

metta
(: Nil (List $ t)) (: Nil (List $ t)) (: Cons (-> $ t (List $ t) (List $ t))) (: Cons (-> $ t (List $ t) (List $ t))) ! (get-type (Cons 1 (Cons 2 Nil))) ! (get-type (Cons 1 (Cons 2 Nil))) ! (get-type (Cons False (Cons True Nil))) ! (get-type (Cons False (Cons True Nil))) ! (get-type (Cons + (Cons - Nil))) ! (get-type (Cons + (Cons - Nil))) ! (get-type (Cons True (Cons 1 Nil))) ! (get-type (Cons True (Cons 1 Nil)))
Run
Let us consider how the type of
(Cons 2 Nil)
is derived. These arguments of
Cons
suppose its type signature to be undergo the following unification:
metta
! (unify (-> $ t (List $ t) (List $ t)) ! (unify (-> $ t (List $ t) (List $ t)) (-> Number (List $ t) $ result) (-> Number (List $ t) $ result) $ result $ result BadType) BadType)
Run
$t
gets bound to
Number
, and the output type
(List $t)
becomes
(List Number)
.
Then, the outer
Cons
in
(Cons 1 (Cons 2 Nil))
receives the arguments of types
Number
and
(List Number)
, which can be simultaneously unified with
$t
and
(List $t)
producing
(List Number)
as the output type once again.
In contrast, the outer
Cons
in
(Cons True (Cons 1 Nil))
receives
Bool
and
(List Number)
. Apparently,

$t
in
(-> $t (List $t) (List $t))
cannot be bound to both
Bool
and
Number
resulting in type error.
Functions can receive arguments of parametric types, and type-checking will help to catch possible mistakes. Consider the following example
sandbox
metta
(: Nil (List $ t)) (: Nil (List $ t)) (: Cons (-> $ t (List $ t) (List $ t))) (: Cons (-> $ t (List $ t) (List $ t))) (: first (-> (List $ t) $ t)) (: first (-> (List $ t) $ t)) (: append (-> (List $ t) (List $ t) (List $ t))) (: append (-> (List $ t) (List $ t) (List $ t))) ! (get-type ! (get-type (+ 1 (+ 1 (first (append (Cons 1 Nil) (first (append (Cons 1 Nil) (Cons 2 Nil))))) (Cons 2 Nil)))))
Run
Copied
Reset
We don't need function bodies for type-checking.
first
returns the first element of
(List $t)
-typed list, and this element should be of
$t
type.
append
concatenates two lists with elements of the same type and produces the list of elements of this type as well. When we start considering a specific expression and unify types of its elements with type signatures of corresponding functions, variables in types get bindings. Apparently, types of
(Cons 1 Nil)
and
(Cons 2 Nil)
are reduced to
(List Number)
. Then,
(append (...) (...))
gets the same type, while the type of
(first (...))
is reduced to
Number
. You can experiment with making the expression badly typed in the code above and see, at which point the error is detected.
Functional programming with types is discussed in more detail in
this tutorial
. However, types in MeTTa are more general than generalized algebraic data types and are similar to dependent types. The use of such advanced types is elaborated in

this tutorial
, in particular, in application to knowledge representation and reasoning.
Metatypes

Peeking into metatypes

In MeTTa, we may need to analyze the structure of atoms themselves. The
tutorial
starts with introducing four kinds of atoms -
Symbol
,
Expression
,
Variable
,
Grounded
. We refer to them as
metatypes
. One can use
get-metatype
to retrieve the metatype of an atom
metta
! (get-metatype 1) ; Grounded ! (get-metatype 1) ; Grounded ! (get-metatype +) ; Grounded ! (get-metatype +) ; Grounded ! (get-metatype (+ 1 2)) ; Expression ! (get-metatype (+ 1 2)) ; Expression ! (get-metatype a) ; Symbol ! (get-metatype a) ; Symbol ! (get-metatype (a b)) ; Expression ! (get-metatype (a b)) ; Expression ! (get-metatype $ x) ; Variable ! (get-metatype $ x) ; Variable
Run
How to process atoms depending on their metatypes is discussed in another tutorial. In this tutorial, we discuss one particular metatype, which is widely utilized in MeTTa to control the order of evaluation. You should have noticed that arguments of some functions are not reduced before the function is called. This is true for
get-type
and
get-metatype
functions. Let us check their type signatures:
metta
! (get-type get-type) ; (-> Atom Atom) ! (get-type get-type) ; (-> Atom Atom) ! (get-type get-metatype) ; (-> Atom Atom) ! (get-type get-metatype) ; (-> Atom Atom)
Run
Here,
Atom
is a supertype for
Symbol
,
Expression
,
Variable

,
Grounded
. While metatypes can appear in ordinary type signatures, they should not be assigned explicitly, e.g.
(: a Expression)
, except for the following special case.
Atom
is treated specially by the interpreter - if a function expects an argument of
Atom
type, this argument is not reduced before passing to the function. This is why, say,
(get-metatype (+ 1 2))
returns
Expression
. It is worth noting that
Atom
as a return result will have no special effect. While
Atom
as the return type could prevent the result from further evaluation, this feature is not implemented in the current version of MeTTa.
Using arguments of
Atom
type is essential for meta-programming and self-reflection in MeTTa. However, it has a lot of other more common uses.
Quoting MeTTa code

We encountered error expressions. These expressions can contain unreduced atoms, because
Error
expects the arguments of
Atom
type:
metta
! (get-type Error) ; (-> Atom Atom ErrorType) ! (get-type Error) ; (-> Atom Atom ErrorType) ! (get-metatype Error) ; just Symbol ! (get-metatype Error) ; just Symbol ! (get-type (Error Foo Boo)) ; ErrorType ! (get-type (Error Foo Boo)) ; ErrorType ! (Error (+ 1 2) (+ 1 +)) ; arguments are not evaluated ! (Error (+ 1 2) (+ 1 +)) ; arguments are not evaluated
Run
Error
is not a grounded atom, it is just a symbol. It doesn't even have defined equalities, so it works just an expression constructor, which prevents its arguments from being evaluated and which has a return type, which can be used to catch errors.
Another very simple constructor from stdlib is
quote
, which is defined just as
(: quote (-> Atom Atom))
. It does nothing except of wrapping its argument and preventing it from being evaluated.
metta
! (get-type quote) ! (get-type quote) ! (quote (+ 1 2)) ! (quote (+ 1 2))

! (get-type if) ! (get-type if)
Run
Some programming languages introduce
quote
as a special symbol known by the interpreter (otherwise its argument would be evaluated). Consequently, any term should be quoted, when we want to avoid evaluating it. However,
quote
is an ordinary symbol in MeTTa. What is specially treated is the
Atom
metatype for arguments. It appears to be convenient not only for extensive work with MeTTa programs in MeTTa itself (for code generation and analysis, automatic programming, meta-programming, genetic programming and such), but also for implementing traditional control statements.
if
under the hood

As was mentioned in the
tutorial
, the
if
statement in MeTTa works much like if-then-else construction in any other language.
if
is not an ordinary function and typically requires a special treatment in interpreters or compilers to avoid evaluation of branches not triggered by the condition.
However, its implementation in MeTTa can be done with the following equalities
metta
(= (if True $ then $ else) $ then) (= (if True $ then $ else) $ then) (= (if False $ then $ else) $ else) (= (if False $ then $ else) $ else)
The trick is to have the type signature with the first argument typed
Bool
, and the next two arguments typed
Atom
. The first argument typed
Bool
can be an expression to evaluate like
(> a 0)
, or a
True
/
False
value. The
Atom
-types arguments
$then
and
$else
will not be evaluated while passing into the
if

function. However, once the
if
-expression has been reduced to either of them, the interpreter will chain its evaluation to obtain the final result.
Consider the following example
sandbox
metta
(: my-if (-> Bool Atom Atom Atom)) (: my-if (-> Bool Atom Atom Atom)) (= (my-if True $ then $ else) $ then) (= (my-if True $ then $ else) $ then) (= (my-if False $ then $ else) $ else) (= (my-if False $ then $ else) $ else) (= (loop) (loop)) (= (loop) (loop)) (= (OK) OK!) (= (OK) OK!) ! (my-if (> 0 1) (loop) (OK)) ! (my-if (> 0 1) (loop) (OK))
Run
Copied
Reset
If you comment out the type definition, then the program will go into an infinite loop trying to evaluate all the arguments of
my-if
. Lazy model of computation could automatically postpone evaluation of
$then
and
$else
expressions until they are not required, but it is not currently implemented.
Can you imagine how a "sequential and" function can be written, which evaluates its second argument, only if the first argument is
True
?
sandbox
metta
(: seq-and (-> ... ... Bool)) (: seq-and (-> ... ... Bool)) (= (seq-and ... ...) ...) (= (seq-and ... ...) ...) (= (seq-and ... ...) ...) (= (seq-and ... ...) ...) (: loop (-> Bool Bool)) (: loop (-> Bool Bool)) ! (seq-and False (loop)) ; should be False ! (seq-and False (loop)) ; should be False ! (seq-and True True) ; should be True ! (seq-and True True) ; should be True
Run
Copied
Reset
Apparently, in the proposed setting, the first argument should be evaluated, so its type should be
Bool
, while the second argument shouldn't be immediately evaluated. What will be the whole solution?
Transforming expressions

One may want to use
Atom
-typed arguments not only for just avoiding computations or quoting expressions, but to modify them before evaluation.
Let us consider a very simple example with swapping the arguments of a function. The

code below will give
-7
as a result
metta
(: swap-arguments-atom (-> Atom Atom)) (: swap-arguments-atom (-> Atom Atom))
(= (swap-arguments-atom ($ op $ arg1 $ arg2)) (= (swap-arguments-atom ($ op $
arg1 $ arg2)) ($ op $ arg2 $ arg1) ($ op $ arg2 $ arg1))) ! (
swap-arguments-atom (- 15 8)) ! (swap-arguments-atom (- 15 8))
Run
At the same time, the same code without typing will not work properly and will
return
[(swap-arguments 7)]
, because
(- 15 8)
will be reduced by the interpreter before passing to the
swap-arguments
and will not be pattern-matched against
($op $arg1 $arg2)
metta
(= (swap-arguments ($ op $ arg1 $ arg2)) (= (swap-arguments ($ op $ arg1 $
arg2)) ($ op $ arg2 $ arg1) ($ op $ arg2 $ arg1))) ! (swap-arguments (- 15
8)) ! (swap-arguments (- 15 8))
Run
One more example of using the
Atom
type is comparing expressions
metta
; `atom-eq` returns True, when arguments are identical ; `atom-eq` returns True,
when arguments are identical ; (can be unified with the same variable) ; (can be
unified with the same variable) (: atom-eq (-> Atom Atom Bool)) (: atom-eq (->
Atom Atom Bool)) (= (atom-eq $ x $ x) True) (= (atom-eq $ x $ x) True) ;
These expressions are identical: ; These expressions are identical: ! (atom-eq (+
1 2) (+ 1 2)) ! (atom-eq (+ 1 2) (+ 1 2)) ; the following will not be
reduced because the expressions are not the same ; the following will not be reduced
because the expressions are not the same ; (even though the result of their
evaluation would be) ; (even though the result of their evaluation would be) ! (
atom-eq 3 (+ 1 2)) ! (atom-eq 3 (+ 1 2))
Run
Controlling pattern matching

Both standard and custom functions in MeTTa can have
Atom
-typed arguments, which will not be reduced before these functions are evaluated.
But we may want to call them on a result of another function call. What is the best
way to do this? Before answering this question, let us consider
match
in more detail.
Type signature of the
match
function

Pattern matching is the core operation in MeTTa, and it is implemented using the
match
function, which locates all atoms in the given Space that match the provided pattern and generates the output pattern.
Let us recall that the
match
function has three arguments:
a grounded atom referencing a Space;
a pattern to be matched against atoms in the Space (query);
an output pattern typically containing variables from the input pattern.
Consider the type of
match
:
metta
! (get-type match) ! (get-type match)
Run
The second and the third arguments are of
Atom
type. Thus, the input and the output pattern are passed to
match
as is, without reduction. Preventing reduction of the input pattern is essentially needed for the possibility to use any pattern for matching. The output pattern is instantiated by
match
and returned, and only then it is evaluated further by the interpreter.
in-and-out behavior of
match

In the following example,
(Green $who)
is evaluated to
True
for
$who
bound to
Tod
due to the corresponding equality.
metta
(Green Sam) (Green Sam) (= (Green Tod) True) (= (Green Tod) True) ! ($ who (Green $ who)) ; (Tod True) ! ($ who (Green $ who)) ; (Tod True) ! (match &self (Green $ who) $ who) ; Sam ! (match &self (Green $ who) $ who) ; Sam
Run
However,
(Green $who)
is not reduced when passed to
match
, and the query returns
Sam
, without utilizing the equality because

(Green Sam)
is added to the Space.
Let us verify that the result of
match
will be evaluated further. In the following example,
match
first finds two entries satisfying the pattern
(Green $who)
and instantiates the output pattern on the base of each of them, but only
(Frog Sam)
is evaluated to
True
on the base of one available equality, while
(Frog Tod)
remains unreduced.
metta
(Green Sam) (Green Sam) (Green Tod) (Green Tod) (= (Frog Sam) True) (= (Frog Sam) True) ! (match &self (Green $ who) (Frog $ who)) ; [True, (Frog Tod)] ! (match &self (Green $ who) (Frog $ who)) ; [True, (Frog Tod)]
Run
We can verify that instantiation of the output pattern happens before its evaluation:
metta
(Green Sam) (Green Sam) (= (Frog Sam) True) (= (Frog Sam) True) ! (match &self (Green $ who) (quote (Frog $ who))) ! (match &self (Green $ who) (quote (Frog $ who)))
Run
Here,
(Green $who)
is matched against
(Green Sam)
,
$who
gets bound to
Sam
, and then it is substituted to the output pattern yielding
(quote (Frog Sam))
, in which
(Frog Sam)
is not reduced further to
True
, because
quote
also expects
Atom
. Thus,
match
can be thought of as transformation of the input pattern to the output pattern. It performs no additional evaluation of patterns by itself.
Returning output patterns with substituted variables before further evaluation is

very convenient for nested queries. Consider the following example:
metta
(Green Sam) (Green Sam) (Likes Sam Emi) (Likes Sam Emi) (Likes Tod Kat) (Likes Tod Kat) ! (match &self (Green $ who) ! (match &self (Green $ who) (match &self (Likes $ who $ x) $ x)) (match &self (Likes $ who $ x) $ x)) ! (match &self (Green $ who) ! (match &self (Green $ who) (match &self (Likes $ boo $ x) $ x)) (match &self (Likes $ boo $ x) $ x)) ! (match &self (Likes $ who $ x) ! (match &self (Likes $ who $ x) (match &self (Green $ x) $ x)) (match &self (Green $ x) $ x)) ! (match &self (Likes $ who $ x) ! (match &self (Likes $ who $ x) (match &self (Green $ boo) $ boo)) (match &self (Green $ boo) $ boo))
Run
The output of the outer query is another query. The inner query is not evaluated by itself, but instantiated as the output of the outer query.
In the first case, $who Sam (Frog Sam $x) Emi gets bound to $who Sam (Frog Sam $x) Emi and the pattern in the second query becomes $who Sam (Frog Sam $x) Emi , which has only one match, so the output is $who Sam (Frog Sam $x) Emi .
In the second case, $who (Likes $boo $x) is not used in the inner query, and there are two results, because the pattern of the second query remains $who (Likes $boo $x) .
In the third case, there are no results, because the outer query produces two results, but neither (Green Emi) (Green Kat) nor (Green Emi) (Green Kat) are in the Space.
In the last case, Sam is returned two times. The outer query returns two results, and although its variables are not used in the inner query, it is evaluated twice.
Patterns are not type-checked

Functions with
Atom
-typed parameters can accept atoms of any other type, including badly typed expressions, which are not supposed to be reduced. As it was mentioned earlier, this behavior can be useful in different situations. Indeed, why couldn't we, say, quote a badly typed expression as an incorrect example?
It should be noted, though, that providing specific types for function parameters and simultaneously indicating that the corresponding arguments should not be reduced could be useful in other cases. Unfortunately, it is currently not possible to provide a specific type and a metatype simultaneously (which is
one of the known issues
).
At the same time,
match
is a very basic function, which should not be restricted in its ability to both accept and return "incorrect" expressions. Thus, one should keep in mind that
match
does not perform type-checking on its arguments, which is intentional and expected.
The following program contains a badly typed expression, which can still be pattern-matched (and
match
can accept a badly typed pattern):
metta

(+ 1 False) (+ 1 False) ! (match &self (+ 1 False) OK) ; OK ! (match &self (+ 1 False) OK) ; OK ! (match &self (+ 1 $ x) $ x) ; False ! (match &self (+ 1 $ x) $ x) ; False
Run
It can be useful to deal with "wrong" MeTTa programs on a meta-level in MeTTa itself, so this behavior of
match
allows us to write code that analyzes badly typed expressions within MeTTa.
Type of
=

MeTTa programs typically contain many equalities. But is there a guarantee that the function will indeed return the declared type? This is achieved by requiring that both parts of equalities are of the same type. Consider the following code:
metta
(: foo (-> Number Bool)) (: foo (-> Number Bool)) (= (foo $ x) (+ $ x 1)) (= (foo $ x) (+ $ x 1)) ! (get-type (foo $ x)) ; Bool ! (get-type (foo $ x)) ; Bool ! (get-type (+ $ x 1)) ; Number ! (get-type (+ $ x 1)) ; Number ! (get-type =) ; (-> $t $t Atom) ! (get-type =) ; (-> $t $t Atom) ! (= (foo $ x) (+ $ x 1)) ; BadType ! (= (foo $ x) (+ $ x 1)) ; BadType
Run
We declared the type of
foo
to be
(-> Number Bool)
. On the base of this definition, the type of
(foo $x)
can be reduced to
Bool
, which is the expected type of its result. However, the type of its body
(+ $x 1)
is reduced to
Number
. If we get the type of
=
, we will see that both its arguments should be of the same type. The result type of
=
is
Atom
, since it is not a function (unless someone adds an equality over equalities, which is permissible). If one tries to "execute" this equality, it will indeed return the type error.
Programs can contain badly typed expressions as we discussed earlier. However, this may permit badly defined functions.
! (pragma! type-check auto)
can be used to enable automatic detection of such errors:
metta
! (pragma! type-check auto) ; () ! (pragma! type-check auto) ; () (: foo (-> Number Bool)) (: foo (-> Number Bool)) (= (foo $ x) (+ $ x 1)) ; BadType (= (foo $ x) (+ $ x 1)) ; BadType

Run
This pragma option turns on type-checking of expressions before adding them to the Space (without evaluation of the expression itself).
let
's evaluate

Sometimes we need to evaluate an expression before passing it to a function, which expects
Atom
-typed arguments. What is the best way to do this?
One trick could be to write a wrapper function like this
metta
```
( = ( call-by-value $ f $ arg ) ( = ( call-by-value $ f $ arg ) ( $ f $ arg )) ( $ f $ arg )) ! ( call-by-value quote ( + 1 2 )) ; (quote 3) ! ( call-by-value quote ( + 1 2 )) ; (quote 3)
```
Run
Arguments of this function are not declared to be of
Atom
type, so they are evaluated before the function is called. Then, the function simply passes its evaluated argument to the given function. However, it is not needed to write such a wrapper function, because there is a more convenient way with the use of operation
let
from stdlib.
let
takes three arguments:
a variable atom (or, more generally, a pattern)
an expression to be evaluated and bound to the variable (or, more generally, matched against the pattern in the first argument)
the output expression (which typically contains a variable to be substituted)
metta
```
! ( let $ x ( + 1 2 ) ( quote $ x )) ; (quote 3) ! ( let $ x ( + 1 2 ) ( quote $ x )) ; (quote 3) ( : Z Nat ) ( : Z Nat ) ! ( get-metatype ( get-type Z )) ; (get-type Z) is Expression ! ( get-metatype ( get-type Z )) ; (get-type Z) is Expression ! ( let $ x ( get-type Z ) ( get-metatype $ x )) ; Nat is Symbol ! ( let $ x ( get-type Z ) ( get-metatype $ x )) ; Nat is Symbol
```
Run
One may also want to evaluate some subexpression before constructing an expression for pattern-matching
metta
```
( = ( age Bob ) 5 ) ( = ( age Bob ) 5 ) ( = ( age Sam ) 8 ) ( = ( age Sam ) 8 ) ( = ( age Ann ) 3 ) ( = ( age Ann ) 3 ) ( = ( age Tom ) 5 ) ( = ( age Tom ) 5 ) ( = ( of-same-age $ who ) ( = ( of-same-age $ who ) ( let $ age ( age $ who ) ( let $ age ( age $ who ) ( match &self ( = ( age $ other ) $ age ) ( match &self ( = ( age $ other ) $ age ) $ other ))) $ other ))) ! ( of-same-age Bob ) ; [Bob, Tom] ! ( of-same-age Bob ) ; [Bob, Tom] ; without `of-same-age`: ; without `of-same-age`: ! ( let $ age ( age Bob ) ! ( let $ age ( age Bob ) ( match &self ( = ( age $ other ) $ age ) ( match &self ( = ( age $ other ) $ age ) $ other )) ; also [Bob, Tom] $ other )) ; also [Bob, Tom] ! ( match &self ( = ( age $ other ) ( age Bob )) ! ( match &self ( = ( age $ other ) ( age Bob )) $ other ) ; does not pattern-match $ other )
```

; does not pattern-match ; evaluating the whole pattern is a bad idea ; evaluating the whole pattern is a bad idea ! (let $ pattern (= (age $ other) (age Bob)) ! (let $ pattern (= (age $ other) (age Bob)) $ pattern) ; [(= 5 5), (= 8 5), (= 5 5), (= 3 5)] $ pattern) ; [(= 5 5), (= 8 5), (= 5 5), (= 3 5)] ! (let $ pattern (= (age $ other) (age Bob)) ! (let $ pattern (= (age $ other) (age Bob)) (match &self $ pattern $ other)) ; does not pattern-match (match &self $ pattern $ other)) ; does not pattern-match
Run
It can be seen that
let
helps to evaluate
(age Bob)
before constructing a pattern for retrieval. However, evaluating the whole pattern is typically a bad idea. That is why patterns in
match
are of
Atom
type, and
let
is used when something should be evaluated beforehand.
As was remarked before,
let
can accept a pattern instead of a single variable. More detailed information on
let
together with other functions from stdlib are provided in
the next tutorial
.
Unit type

Unit
is a type that has exactly one possible value
unit
serving as a return value for functions, which return "nothing". However, from the type-theoretic point of view, mappings to the empty set do not exist (they are non-constructive), while mappings to the one-element set do exist, and returning the only element of this set yields zero information, that is, "nothing". This is equivalent to
void
in such imperative languages as C++.
In MeTTa, the empty expression
()
is used for the unit value, which is the only instance of the type
(->)
. A function, which doesn't return anything meaningful but which is still supposed to be a valid function, should return
()
unless a custom unit type is defined for it.
In practice, this
()
value is used as the return type for grounded functions with side effects (unless

these side effects are not described in a special way, e.g., with monads). For example, the function
add-atom
adds an atom to the Space, and returns
()
.
When it is necessary to execute such a side-effect function and then to return some value, or to chain it with subsequent execution of another side-effect function, it is convenient to use the following construction based on
let
:
(let () (side-effect-function) (evaluate-next))
. If
(side-effect-function)
returns
()
, it is matched with the pattern
()
in
let
-expression (one can use a variable instead of
()
as well), and then
(evaluate-next)
is executed.
Let us consider a simple knowledge base for a personal assistant system. The knowledge base contains information about the tasks the user is supposed to do. A new atom in this context would be a new task.
metta
```
( = ( message-to-user $ task ) ( = ( message-to-user $ task ) ( Today you have $ task )) ( Today you have $ task )) ( = ( add-task-and-notify $ task ) ( = ( add-task-and-notify $ task ) ( let () ( add-atom &self ( TASK $ task )) ( let () ( add-atom &self ( TASK $ task )) ( message-to-user $ task )) ( message-to-user $ task )) ) ) ! ( get-type add-atom ) ; (-> hyperon::space::DynSpace Atom (->)) ! ( get-type add-atom ) ; (-> hyperon::space::DynSpace Atom (->)) ! ( add-task-and-notify ( Something to do )) ! ( add-task-and-notify ( Something to do )) ! ( match &self ( TASK $ t ) $ t ) # ( Somthing to do ) ! ( match &self ( TASK $ t ) $ t ) # ( Somthing to do )
```
Run
The
add-task-and-notify
function adds a
$task
atom into the current Space using the
add-atom
function and then calls another function which returns a message to notify the user about the new task. Please, notice the type signature of
add-atom
.
Standard Library Overview

In this section we will look at the main functions of the standard library, which are part of the standard distribution of MeTTa.
Table of Contents

Basic grounded functions

Arithmetic operators and
Number
type

Arithmetic operations in MeTTa are grounded functions and use the prefix notation where the operator comes before the operands. MeTTa arithmetic works with atoms of
Number
type, which can store floating-point numbers as well as integers under the hood, and you can mix them in your calculations. The type of binary arithmetic operations is
(-> Number Number Number)
.
metta
; Addition ; Addition ! (+ 1 3) ; 4 ! (+ 1 3) ; 4 ; Subtraction ; Subtraction ! (- 6 2.2) ; 3.8 ! (- 6 2.2) ; 3.8 ; Multiplication ; Multiplication ! (* 7.3 9) ; 65.7 ! (* 7.3 9) ; 65.7 ; Division ; Division ! (/ 25 5) ; 5 or 5.0 ! (/ 25 5) ; 5 or 5.0 ; Modulus ; Modulus ! (% 24 5) ; 4 ! (% 24 5) ; 4
Run
In the current implementation arithmetic operations support only two numerical arguments, expressions with more than two arguments like
!(+ 1 2 3 4)
will result in a type error (
IncorrectNumberOfArguments
). One should use an explicit nested expression in that case
metta
! (+ 1 (+ 2 (+ 3 4))) ; 10 ! (+ 1 (+ 2 (+ 3 4))) ; 10 ! (- 8 (/ 6.4 4)) ; 6.4 ! (- 8 (/ 6.4 4)) ; 6.4
Run
Numbers in MeTTa are presented as grounded atoms with the predefined
Number
type. Evaluation of ill-typed expressions produces an error expression. Notice, however, that arithmetic expressions with atoms of
%Undefined%
type will not be reduced.
metta
! (+ 2 S) ; (+ 2 S) ! (+ 2 S) ; (+ 2 S) ! (+ 2 "8") ; BadType ! (+ 2 "8") ; BadType
Run

Other common mathematical operations like
sqr
,
sqrt
,
abs
,
pow
,
min
,
max
,
log2
,
ln
, etc. are not included in the standard library as grounded symbols at the moment. But they can be
imported from Python directly
.
Comparison operations

Comparison operations implemented in stdlib are also grounded operations. There are four operations
<
,
>
,
<=
,
>=
of
(-> Number Number Bool)
type.
metta
```
; Less than ; Less than ! ( < 1 3 ) ! ( < 1 3 )  ; Greater than ; Greater than ! ( > 3 2 ) ! ( > 3 2 )  ; Less than or equal to ; Less than or equal to ! ( <= 5 6.2 ) ! ( <= 5 6.2 )  ; Greater than or equal to ; Greater than or equal to ! ( >= 4 ( + 2 ( * 3 5 ))) ! ( >= 4 ( + 2 ( * 3 5 )))
```
Run
Once again, passing ordinary symbols to grounded operations will not cause errors, and the expression simply remains unreduced, if it type-checks. Thus, it is generally a good practice to ensure the types of atoms being compared are what the comparison operators expect to prevent unexpected results or errors.
metta
```
! ( > $ x ( + 8 2 )) ; Inner expression is reduced, but the outer is not ! ( > $ x ( + 8 2 )) ; Inner expression is reduced, but the outer is not ! ( >= 4 ( + Q 2 )) ; Reduction stops in the inner expression ! ( >= 4 ( + Q 2 )) ; Reduction stops in the inner expression ( : R CustomType ) ( : R CustomType ) ! ( >= 4 R ) ; BadType ! ( >= 4 R ) ; BadType
```

Run
The
==
operation is implemented to work with both grounded and symbol atoms and expressions (while remaining a grounded operation). Its type is
(-> $t $t Bool)
. Its arguments are evaluated before executing the operation itself.
metta
! (== 4 (+ 2 2)) ; True ! (== 4 (+ 2 2)) ; True ! (== "This is a string" "Just a string") ; False ! (== "This is a string" "Just a string") ; False ! (== (A B) (A B)) ; True ! (== (A B) (A B)) ; True ! (== (A B) (A (B C))) ; False ! (== (A B) (A (B C))) ; False
Run
Unlike
<
or
>
,
==
will not remain unreduced if one of its arguments is grounded, while another is not. Instead, it will return
False
if the expression is well-typed.
metta
! (== 4 (+ Q 2)) ; False ! (== 4 (+ Q 2)) ; False (: R CustomType) (: R CustomType) ! (== 4 R) ; BadType ! (== 4 R) ; BadType
Run
Logical operations and
Bool
type

Logical operations in MeTTa can be (and with some build options are) implemented purely symbolically. However, the Python version of stdlib contains their grounded implementation for better interoperability with Python. In particular, numeric comparison operations directly execute corresponding operations in Python and wrap the resulting
bool
value into a grounded atom. The grounded implementation is intended for subsymbolic and purely functional processing, while custom logic systems for reasoning are supposed to be implemented symbolically in MeTTa itself.
Logical operations in stdlib deal with
True
and
False
values of
Bool
type, and have signatures
(-> Bool Bool)
and
(-> Bool Bool Bool)

for unary and binary cases.
metta
; Test if both the given expressions are True ; Test if both the given expressions are True ! (and (> 4 2) (== "This is a string" "Just a string")) ; False ! (and (> 4 2) (== "This is a string" "Just a string")) ; False ; Test if any of the given expressions is True ; Test if any of the given expressions is True ! (or (> 4 2) (== "This is a string" "Just a string")) ; True ! (or (> 4 2) (== "This is a string" "Just a string")) ; True ; Negates the result of a given Bool value ; Negates the result of a given Bool value ! (not (== 5 5)) ; False ! (not (== 5 5)) ; False ! (not (and (> 4 2) (< 4 3))) ; True ! (not (and (> 4 2) (< 4 3))) ; True
Run
Console output and debugging

All values obtained during evaluation of the MeTTa program or script are collected and returned. The whole program can be treated as a function. If a stand-alone program is executed via
a command-line runner
or
REPL
these results are printed at the end. This printing will not happen if MeTTa is used via
its API
.
However, MeTTa has two functions to send information to the console output:
println!
and
trace!
. They can be used by developers for displaying messages and logging information during the evaluation process, in particular, for debugging purposes.
Print a line

The
println!
function is used to print a line of text to the console. Its type signature is
(-> %Undefined% (->))
.
The function accepts only a single argument, but multiple values can be printed by enclosing them within parentheses to form a single atom:
metta
! (println! "This is a string") ! (println! "This is a string") ! (println! ($ v1 "string" 5)) ! (println! ($ v1 "string" 5))
Run
Note that
println!
returns the
unit
value
()
. Beside printing to stdout, the program will return two units due to

println!
evaluation.
The argument of
println!
is evaluated before
println!
is called (its type is not
Atom
but
%Undefined%
), so the following code
metta
(Parent Bob Ann) (Parent Bob Ann) ! (match &self (Parent Bob Ann) (Ann is Bob`s child )) ! ( match &self ( Parent Bob Ann ) ( Ann is Bob`s child)) ! (println! (match &self (Parent Bob Ann) (Bob is Ann`s parent ))) ! ( println! ( match &self ( Parent Bob Ann ) ( Bob is Ann`s parent)))
Run
will print
(Bob is Ann's parent)
to stdout. Note that this result is printed before all the evaluation results (starting with the
match
expressions) are returned.
Trace log

trace!
accepts two arguments, the first is the atom to print, and the second is the atom to return. Both are evaluated before passing to
trace!
, which type is
(-> %Undefined% $a $a)
, meaning that the reduced type of the whole
trace!
expression is the same as the reduced type of the second argument:
metta
! (get-type (trace! (Expecting 3) (+ 1 2))) ; Number ! (get-type (trace! (Expecting 3) (+ 1 2))) ; Number
Run
trace!
can be considered as a syntactic sugar for the following construction using
println!
and
let
(see
this section
of the tutorial for more detail):
metta
(: my-trace (-> %Undefined% $ a $ a)) (: my-trace (-> %Undefined% $ a $ a)) (= (my-trace $ out $ res) (= (my-trace $ out $ res) (let () (println! $ out) $ res)) (let () (println! $ out) $ res)) ! (my-trace (Expecting 3) (+ 1 2

)) ! (my-trace (Expecting 3) (+ 1 2))
Run
It can be used as a debugging tool that allows printing out a message to the terminal, along with valuating an atom.
metta
(Parent Bob Ann) (Parent Bob Ann) ! (trace! "Who is Anna`s parent?" ; print this expression ! ( trace! "Who is Anna`s parent?" ; print this expression (match &self (Parent $ x Ann) (match &self (Parent $ x Ann) ($ x is Ann`s parent ))) ; return the result of this expression ( $ x is Ann`s parent))) ; return the result of this expression ! (trace! "Who is Bob`s child?" ; print this expression ! ( trace! "Who is Bob`s child?" ; print this expression (match &self (Parent Bob $ x) (match &self (Parent Bob $ x) ($ x is Bob`s child ))) ; return the result of this expression ( $ x is Bob`s child))) ; return the result of this expression
Run
The first argument does not have to be a pure string, which makes
trace!
work fine on its own
metta
(Parent Bob Ann) (Parent Bob Ann) ! (trace! ((Expected: (Bob is Ann`s parent )) ! ( trace! (( Expected: ( Bob is Ann`s parent)) (Got: (match &self (Parent $ x Ann) ($ x is Ann`s parent ))) ( Got: ( match &self ( Parent $ x Ann ) ( $ x is Ann`s parent))))) ()) ())
Run
Quote

Quotation was
already introduced
as a tool for evaluation control. Let us recap that
quote
is just a symbol with
(-> Atom Atom)
type without equalities (i.e., a constructor). In some versions of MeTTa and its stdlib,
quote
can be defined as
(= (quote $atom) NotReducible)
, where the symbol
NotReducible
explicitly tells the interpreter that the expression should not be reduced.
The following is the basic example of the effect of
quote
:
metta
(Fruit apple) (Fruit apple) (= (fruit $ x) (= (fruit $ x) (match &self (Fruit $ x) $ x)) (match &self (Fruit $ x) $ x)) ! (fruit $ x) ; apple ! (fruit $ x) ; apple ! (quote (fruit $ x)) ; (quote (fruit $x)) ! (quote (fruit $ x)) ; (quote (fruit $x))
Run
There is a useful combination of
trace!

,
quote
, and
let
for printing an expression together with its evaluation result, which is then returned.
metta
(: trace-eval (-> Atom Atom)) (: trace-eval (-> Atom Atom)) (= (trace-eval $ expr) (= (trace-eval $ expr) (let $ result $ expr (let $ result $ expr (trace! (EVAL: (quote $ expr) --> $ result) (trace! (EVAL: (quote $ expr) --> $ result) $ result))) $ result))) (Fruit apple) (Fruit apple) (= (fruit $ x) (= (fruit $ x) (match &self (Fruit $ x) $ x)) (match &self (Fruit $ x) $ x)) ; (EVAL: (quote (fruit $x)) --> apple) is printed to stdout ; (EVAL: (quote (fruit $x)) --> apple) is printed to stdout ! (Overall result is (trace-eval (fruit $ x))) ; (Overall result is apple) ! (Overall result is (trace-eval (fruit $ x))) ; (Overall result is apple)
Run
In this code,
trace-eval
accepts
$expr
of
Atom
type, so it is not evaluated before getting to
trace-eval
.
(let $result $expr ...)
stores the result of evaluation of
$expr
into
$result
, and then prints both of them using
trace!
(
(quote $expr)
is used to avoid reduction of
$expr
before passing to
trace!
) and returns
$result
. The latter allows wrapping
trace-eval
into other expressions, which results in the behavior, which would take place without such wrapping, except for additional console output.
Another pattern of using
trace!
with
quote
and

let
is to add tracing to the function itself. We first calculate the result (if needed), and then use
trace!
to print some debugging information and return the result:
metta
(= (add-bin $ x) (= (add-bin $ x) (let $ r (+ $ x 1) (let $ r (+ $ x 1) (trace! (quote ((add-bin $ x) is $ r)) (trace! (quote ((add-bin $ x) is $ r)) $ r))) $ r))) (= (add-bin $ x) (= (add-bin $ x) (trace! (quote ((add-bin $ x) is $ x)) (trace! (quote ((add-bin $ x) is $ x)) $ x)) $ x)) ; (quote ((add-bin 1) is 1)) and (quote ((add-bin 1) is 2)) will be printed ; (quote ((add-bin 1) is 1)) and (quote ((add-bin 1) is 2)) will be printed ! (add-bin 1) ; [1, 2] ! (add-bin 1) ; [1, 2]
Run
Without quotation an atom such as
(add-bin $x)
evaluated from
trace!
would result in an infinite loop, but
quote
prevents the wrapped atom from being interpreted.
In the following code
(test 1)
would be evaluated from
trace!
and would result in an infinite loop
metta
(= (test 1) (trace! (test 1) 1)) (= (test 1) (trace! (test 1) 1)) (= (test 1) (trace! (test 0) 0)) (= (test 1) (trace! (test 0) 0)) ! (test 1) ! (test 1)
Asserts

MeTTa has a couple of assert operations that allow a program to check if a certain condition is true and return an error-expression if it is not.
assertEqual
compares (sets of) results of evaluation of two expressions. Its type is
(-> Atom Atom Atom)
, so it interprets expressions internally and can compare erroneous expressions. If sets of results are equal, it outputs the unit value
()
.
metta
(Parent Bob Ann) (Parent Bob Ann) ! (assertEqual ! (assertEqual (match &self (Parent $ x Ann) $ x) (match &self (Parent $ x Ann) $ x) (unify (Parent $ x Ann) (Parent Bob $ y) $ x Failed)) ; () (unify (Parent $ x Ann) (Parent Bob $ y) $ x Failed)) ; () ! (assertEqual (+ 1 2) 3) ; () ! (assertEqual (+ 1 2) 3) ; () ! (assertEqual (+ 1 2) (+ 1 4)) ; Error-expression ! (assertEqual (+ 1 2) (+ 1 4)) ; Error-expression
Run
While

assertEqual
is convenient when we have two expressions to be reduced to the same result, it is quite common that we want to check if the evaluated expression has a very specific result. Imagine the situation when one wants to be sure that some expression, say
(+ 1 x)
, is
not
reduced. It will make no sense to use
(assertEqual (+ 1 x) (+ 1 x))
.
Also, if the result of evaluation is nondeterministic, and the set of supposed outcomes is known, one would need to turn this set into a nondeterministic result as well in order to use
assertEqual
. It can be done with
superpose
, but both issues are covered by the following assert function.
assertEqualToResult
has the same type as
assertEqual
, namely
(-> Atom Atom Atom)
, and it evaluates the first expression. However, it doesn't evaluate the second expression, but considers it a set of expected results of the first expression.
metta
(Parent Bob Ann) (Parent Bob Ann) (Parent Pam Ann) (Parent Pam Ann) ! (assertEqualToResult ! (assertEqualToResult (match &self (Parent $ x Ann) $ x) (match &self (Parent $ x Ann) $ x) (Bob Pam)) ; () (Bob Pam)) ; () (= (bin) 0) (= (bin) 0) (= (bin) 1) (= (bin) 1) ! (assertEqualToResult (bin) (0 1)) ; () ! (assertEqualToResult (bin) (0 1)) ; () ! (assertEqualToResult (+ 1 2) (3)) ; () ! (assertEqualToResult (+ 1 2) (3)) ; () ! (assertEqualToResult ! (assertEqualToResult (+ 1 untyped-symbol) (+ 1 untyped-symbol) ((+ 1 untyped-symbol))) ; () ((+ 1 untyped-symbol))) ; () ! (assertEqualToResult (+ 1 2) ((+ 1 2))) ; Error ! (assertEqualToResult (+ 1 2) ((+ 1 2))) ; Error
Run
Let us notice a few things:
We have to take the result into brackets, e.g., (assertEqualToResult (+ 1 2) (3)) (assertEqual (+ 1 2) 3) assertEqualToResult vs (assertEqualToResult (+ 1 2) (3)) (assertEqual (+ 1 2) 3) assertEqualToResult , because the second argument of (assertEqualToResult (+ 1 2) (3)) (assertEqual (+ 1 2) 3) assertEqualToResult is a set of results even if this set contains one element.
As a consequence, a non-reducible expression also gets additional brackets as the second argument, e.g., ((+ 1 untyped-symbol)) . It is also a one-element set of the results.
The second argument is indeed not evaluated. The last assert yields an error, because (+ 1 2) 3 3 (+ 1 2) is reduced to (+ 1 2) 3 3 (+ 1 2) . Notice (+ 1 2) 3 3 (+ 1 2) as what we got instead of expected (for the sake of the example) (+ 1 2) 3 3 (+ 1 2) .
Handling nondeterministic results

Superpose

In previous tutorials we saw that
match
along with any other function can return multiple (nondeterministic) as well as empty results. If you need to get a nondeterministic result explicitly, use the
superpose
function, which turns a tuple into a nondeterministic result. It is an stdlib function of
(-> Expression Atom)
type.
However, it is typically recommended to avoid using it. For example, in the following program
metta
```
( = ( bin ) 0 ) ( = ( bin ) 0 ) ( = ( bin ) 1 ) ( = ( bin ) 1 ) ( = ( bin2 ) ( superpose ( 0 1 ))) ( = ( bin2 ) ( superpose ( 0 1 ))) ! ( bin ) ; [0, 1] ! ( bin ) ; [0, 1] ! ( bin2 ) ; [0, 1] ! ( bin2 ) ; [0, 1]
```
Run
bin
and
bin2
do similar job. However,
bin
is evaluated using one equality query, while
bin2
requires additional evaluation of
superpose
. Also, one may argue that
bin
is more modular and more suitable for meta-programming and evaluation control.
One may want to use
superpose
to execute several operations. However, the order of execution is not guaranteed. And again, one can try thinking about writing multiple equalities for a function, inside which
superpose
seems to be suitable.
However,
superpose
can still be convenient in some cases. For example, one can pass nondeterministic expressions to any function (both grounded and symbolic, built-in and custom) and get multiple results. In the following example, writing a nondeterministic function returning
3
,
4
,
5
would be inconvenient:

metta
! (+ 2 (superpose (3 4 5))) ; [5, 6, 7] ! (+ 2 (superpose (3 4 5))) ; [5, 6, 7]
Run
Here, nondeterminism works like a map over a set of elements.
Another example, where using
superpose
explicitly is useful is for checking a set of nondeterministic results with
assertEqual
, when both arguments still require evaluation (so
assertEqualToResult
is not convenient to apply). In the following example, we want to check that we didn't forget any equality for
(color)
, but we may not be interested what exact value they are reduced to (i.e., whether
(ikb)
is reduced to
international-klein-blue
or something else).
metta
(= (ikb) international-klein-blue) (= (ikb) international-klein-blue) (= (color) green) (= (color) green) (= (color) yellow) (= (color) yellow) (= (color) (ikb)) (= (color) (ikb)) ! (assertEqual ! (assertEqual (match &self (= (color) $ x) $ x) (match &self (= (color) $ x) $ x) (superpose ((ikb) yellow green))) ; () (superpose ((ikb) yellow green))) ; () ! (assertEqualToResult ! (assertEqualToResult (match &self (= (color) $ x) $ x) (match &self (= (color) $ x) $ x) ((ikb) yellow green)) ; Error ((ikb) yellow green)) ; Error
Run
Empty

As mentioned above, in MeTTa, functions can return empty results. This is a natural consequence on the evaluation semantics based on queries, which can find no matches. Sometimes, we may want to force a function to "return" an empty result to abort a certain evaluation branch, or to explicitly represent it to analyze this behavior on the meta-level.
(superpose ())
will exactly return the empty set of results. However, stdlib provide
(empty)
function to do the same in a clearer and stable way. Some versions may also use
Empty
as a symbol to inform the interpreter about the empty result, which may differ on some level from calling a grounded function, which really returns an empty set.
(empty)
is supported more widely at the moment, so we use it here.
(empty)
could be useful in the construction of the asserts
(assertEqual (...) (empty))
, but
(assertEqualToResult (...) ())

can also work.
metta
(Parent Bob Ann) (Parent Bob Ann) ! (assertEqual ! (assertEqual (match &self (Parent Tom $ x) $ x) (match &self (Parent Tom $ x) $ x) (empty)) ; () (empty)) ; () ! (assertEqualToResult ! (assertEqualToResult (match &self (Parent Tom $ x) $ x) (match &self (Parent Tom $ x) $ x) ()) ; () ()) ; ()
Run
Since expressions without suitable equalities remain unreduced in MeTTa,
(empty)
can be used to alter this behavior, when desirable, e.g.
metta
(= (eq $ x $ x) True) (= (eq $ x $ x) True) ! (eq a b) ; (eq a b) ! (eq a b) ; (eq a b) (= (eq $ x $ y) (empty)) (= (eq $ x $ y) (empty)) ! (eq a b) ; no result ! (eq a b) ; no result
Run
(empty)
can be used to turn a total function such as
if
or
unify
into a partial function, when we have no behavior for the else-branch, and we don't want the expression to remain unreduced.
Let us note that there is some convention in how the interpreter processes empty results. If the result of
match
for equality query is empty, the interpreter doesn't reduce the given expression (it transforms the empty result of such queries to
NotReducible
), but if a grounded function returns the empty result, it is treated as partial. When a grounded function application is not reduced, e.g.
(+ 1 undefined-symbol)
, because the function returns not the empty result, but
NotReducible
. This behavior may be refined in the future, but the possibility to have both types of behavior (a partial function is not reduced and evaluation continues or it returns no result stopping further evaluation) will be supported.
From nondeterministic viewpoint,
(empty)
removes an evaluation branch. If we consider all the results as a collection,
(empty)
can be used for its filtering. In the following program,
(color)
and
(fruit)
produce nondeterministic "collections" of colors and fruits correspondingly, while
filter-prefer
is a partially defined id function, which can be used to filter out these collections.
metta
(= (color) red) (= (color) red) (= (color) green) (= (color) green)

(= (color) blue) (= (color) blue) (= (fruit) apple) (= (fruit) apple) (= (fruit) banana) (= (fruit) banana) (= (fruit) mango) (= (fruit) mango) (= (filter-prefer blue) blue) (= (filter-prefer blue) blue) (= (filter-prefer banana) banana) (= (filter-prefer banana) banana) (= (filter-prefer mango) mango) (= (filter-prefer mango) mango) (= (filter-prefer $ x) (empty)) (= (filter-prefer $ x) (empty)) ! (filter-prefer (color)) ; [blue] ! (filter-prefer (color)) ; [blue] ! (filter-prefer (fruit)) ; [mango, banana] ! (filter-prefer (fruit)) ; [mango, banana]
Run
In case of recursion,
(empty)
can prune branches, which don't satisfy some conditions as shown in
this example
.
Collapse

Nondeterminism is an efficient way to map and filter sets of elements as well as to perform search. However, nondeterministic branches do not "see" each other, while we may want to get the extreme element or just to count them (or, more generally, fold over them). That is, we may need to collect the results in one evaluation branch.
Reverse operation to
superpose
is
collapse
, which has the type
(-> Atom Expression)
. It converts a nondeterministic result into a tuple.
collapse
is a grounded function, which runs the interpreter on the given atom and wraps the returned results into an expression.
metta
(= (color) red) (= (color) red) (= (color) green) (= (color) green) (= (color) blue) (= (color) blue) ! (color) ; three results: [blue, red, green] ! (color) ; three results: [blue, red, green] ! (collapse (color)) ; one result: [(blue red green)] ! (collapse (color)) ; one result: [(blue red green)]
Run
Here we've got a nondeterministic result
[blue, red, green]
from the
color
function and converted it into one tuple
[(blue red green)]
using
collapse
.
The
superpose
function reverts the
collapse

result
metta
(= (color) green) (= (color) green) (= (color) yellow) (= (color) yellow) (= (color) red) (= (color) red) ! (color) ; [green, yellow, red] ! (color) ; [green, yellow, red] ! (collapse (color)) ; [(green yellow red)] ! (collapse (color)) ; [(green yellow red)] ! (let $ x (collapse (color)) ! (let $ x (collapse (color)) (superpose $ x)) ; [green, yellow, red] (superpose $ x)) ; [green, yellow, red] ! (superpose (1 2 3)) ; [1, 2, 3] ! (superpose (1 2 3)) ; [1, 2, 3] ! (collapse (superpose (1 2 3))) ! (collapse (superpose (1 2 3))) ! (let $ x (superpose (1 2 3)) ; [(1 2 3)] ! (let $ x (superpose (1 2 3)) ; [(1 2 3)] (collapse $ x)) ; [(1), (2), (3)] (collapse $ x)) ; [(1), (2), (3)]
Run
The color function gives the nondeterministic result
[green, yellow, red]
(the order of colors may vary). The
collapse
function converts it into a tuple
[(green yellow red)]
. And finally the
superpose
function in
let
converts a tuple back into the nondeterministic result
[red, green, yellow]
. The order of colors may change again due to nondeterminism.
Note that we cannot call
collapse
inside
superpose
, because
collapse
will not be executed before passing to
superpose
and will be considered as a part of the input tuple. In contrary, we cannot call
superpose
outside
collapse
, because it will cause
collapse
to be called separately for each nondeterministic branch produced by superpose instead of collecting these branches inside
collapse
.
Working with spaces

Space API

Spaces can have different implementations, but should satisfy a certain API. This API includes pattern-matching (or unification) functionality.

match
is an stdlib function, which calls a corresponding API function of the given space, which can be different from the program space.
Let us recap that the type of
match
is
(-> hyperon::space::DynSpace Atom Atom %Undefined%)
. The first argument is a space (or, more precisely, a grounded atom referring to a space) satisfying the Space API. The second argument is the input pattern to be unified with expressions in the space, and the third argument is the output pattern, which is instantiated for every found match.
match
can produce any number of results starting with zero, which are treated nondeterministically.
The basic use of
match
was already covered before, while its use with custom spaces will be described in other tutorials, since these spaces are not the part of stdlib. However, the Space API includes additional components, which are utilized by such stdlib functions as
add-atom
and
remove-atom
.
Adding atoms

The content of spaces can be not only defined statically in MeTTa scripts, but can also be modified at runtime by programs residing in the same or other spaces.
The function
add-atom
adds an atom into the Space. Its type is
(-> hyperon::space::DynSpace Atom (->))
. The first argument is an atom referring some Space, to which an atom provided as the second argument will be added. Since the type of the second argument is
Atom
, the added atom is added as is without reduction.
In the following program,
add-foo-eq
is a function, which adds an equality for
foo
to the program space whenever called. Then, it is checked that the expressions are added to the space without reduction.
metta
(: add-foo-eq (-> Atom (->))) (: add-foo-eq (-> Atom (->))) (= (add-foo-eq $ x) (= (add-foo-eq $ x) (add-atom &self (= (foo) $ x))) (add-atom &self (= (foo) $ x))) ! (foo) ; (foo) - not reduced ! (foo) ; (foo) - not reduced ! (add-foo-eq (+ 1 2)) ; () - OK ! (add-foo-eq (+ 1 2)) ; () - OK ! (add-foo-eq (+ 3 4)) ; () - OK ! (add-foo-eq (+ 3 4)) ; () - OK ! (foo) ; [3, 7] ! (foo) ; [3, 7] ! (match &self (= (foo) $ x) ! (match &self (= (foo) $ x) (quote $ x)) ; [(quote (+ 1 2)), (quote (+ 3 4))] (quote $ x)) ; [(quote (+ 1 2)), (quote (+ 3 4))]

Run
If it is desirable to add a reduced atom without additional wrappers (e.g., like
add-foo-eq
but without
Atom
type for the argument), then
add-reduct
can be used:
metta
! (add-reduct &self (= (foo) (+ 3 4))) ; () ! (add-reduct &self (= (foo) (+ 3 4))) ; () ! (foo) ; 7 ! (foo) ; 7 ! (match &self (= (foo) $ x) ! (match &self (= (foo) $ x) (quote $ x)) ; (quote 7) (quote $ x)) ; (quote 7)
Run
Removing atoms

The function
remove-atom
removes an atom from the AtomSpace without reducing it. Its type is
(-> hyperon::space::DynSpace Atom (->))
.
The first argument is a reference to the space from which the Atom needs to be removed, the second is the atom to be removed. Notice that if the given atom is not in the space,
remove-atom
currently neither raises a error nor returns the empty result.
metta
(Atom to remove) (Atom to remove) ! (match &self (Atom to remove) "Atom exists") ; "Atom exists" ! (match &self (Atom to remove) "Atom exists") ; "Atom exists" ! (remove-atom &self (Atom to remove)) ; () ! (remove-atom &self (Atom to remove)) ; () ! (match &self (Atom to remove) "Unexpected") ; nothing ! (match &self (Atom to remove) "Unexpected") ; nothing ! (remove-atom &self (Atom to remove)) ; () ! (remove-atom &self (Atom to remove)) ; ()
Run
Combination of
remove-atom
and
add-atom
can be used for
graph rewriting
. Consider the following example.
metta
(link A B) (link A B) (link B C) (link B C) (link C A) (link C A) (link C E) (link C E) ! (match &self (, (link $ x $ y) ! (match &self (, (link $ x $ y) (link $ y $ z) (link $ y $ z) (link $ z $ x)) (link $ z $ x)) (let () (remove-atom &self (link $ x $ y)) (let () (remove-atom &self (link $ x $ y)) (add-atom &self (link $ y $ x))) (add-atom &self (link $ y $ x)))) ; [(), (), ()]) ; [(), (), ()] ! (match &self (link $ x $ y) ! (match &self (link $ x $ y) (link $ x $ y)) ; [(link A C), (link C B), (link B A), (link C E)] (link $ x $ y)) ; [(link A C), (link C B), (link B A), (link C E)]
Run

Here, we find entries
(link _ _)
, which form three-element loops, and revert the direction of links in them. Let us note that
match
returns three unit results, because the loop can start from any of such entries. All of them are reverted (only
(link C E)
remains unchanged). Also, in the current implementation,
match
first finds all the matches, and then instantiates the output pattern with them, which is evaluated outside
match
. If
remove-atom
and
add-atom
would be executed right away for each found matching, the condition of circular links would be broken after the first rewrite. This behavior can be space-specific, and is not a part of MeTTa specification at the moment. This can be changed in the future.
New spaces

It is possible to create other spaces with the use of
new-space
function from stdlib. Its type is
(-> hyperon::space::DynSpace)
, so it has no arguments and returns a fresh space. Creating new spaces can be useful to keep the program space cleaner, or to simplify queries.
If we just run
(new-space)
like this
metta
```
! ( new-space ) ! ( new-space )
```
Run
we will get something like
GroundingSpace-0x10703b398
as a textual representation space atom. But how can we refer to this space in other parts of the program? Notice that the following code will not work as desired
metta
```
( = ( get-space ) ( new-space )) ( = ( get-space ) ( new-space )) ! ( add-atom ( get-space ) ( Parent Bob Ann )) ; () ! ( add-atom ( get-space ) ( Parent Bob Ann )) ; () ! ( match ( get-space ) ( Parent $ x $ y ) ( $ x $ y )) ; nothing ! ( match ( get-space ) ( Parent $ x $ y ) ( $ x $ y )) ; nothing
```
Run
because
(get-space)
will create a brand new space each time.
One workaround for this issue in a functional programming style is to wrap the whole program into a function, which accepts a space as an input and passes it to

subfunctions, which need it:
metta
(= (main $ space) (= (main $ space) (let () (add-atom $ space (Parent Bob Ann)) (let () (add-atom $ space (Parent Bob Ann)) (match $ space (Parent $ x $ y) ($ x $ y)) (match $ space (Parent $ x $ y) ($ x $ y)))))) ! (main (new-space)) ; (Bob Ann) ! (main (new-space)) ; (Bob Ann)
Run
This approach has its own merits. However, a more direct fix for
(= (get-space) (new-space))
would be just to evaluate
(new-space)
before adding it to the program:
metta
! (add-reduct &self (= (get-space) (new-space))) ; () ! (add-reduct &self (= (get-space) (new-space))) ; () ! (add-atom (get-space) (Parent Bob Ann)) ; () ! (add-atom (get-space) (Parent Bob Ann)) ; () ! (get-space) ; GroundingSpace-addr ! (get-space) ; GroundingSpace-addr ! (match (get-space) (Parent $ x $ y) ($ x $ y)) ; (Bob Ann) ! (match (get-space) (Parent $ x $ y) ($ x $ y)) ; (Bob Ann)
Run
That is,
(new-space)
is evaluated to a grounded atom, which wraps a newly created space. Other elements of
(= (get-space) (new-space))
are not reduced. Instead of
add-reduct
, one could use the following more explicit code
metta
! (let $ space (new-space) ! (let $ space (new-space) (add-atom &self (= (get-space) $ space))) (add-atom &self (= (get-space) $ space)))
which also ensured that nothing is reduced except
(new-space)
.
Creating tokens

Why can't we refer to the grounded atom, which wraps the created space? Indeed, we can represent such grounded atoms as numbers or operations over them in the code. And what is about
&self
?
In fact, they are turned into atoms from their textual representation by the parser, which knows a mapping from textual tokens (defined with the use of regular expressions) to constructors of corresponding grounded atom. Basically,
&self
is replaced with the grounded atom wrapping the program space by the parser before it gets inside the interpreter.
Parsing is explained in more detail in
another tutorial
, while here we focus on the stdlib function

bind!
.
bind!
registers a new token which is replaced with an atom during the parsing of the rest of the program. Its type is
(-> Symbol %Undefined% (->))
.
The first argument has type
Symbol
, so technically we can use any valid symbol as the token name, but conventionally the token should start with
&
, when it is bound to a custom grounded atom, to distinguish it from symbols. The second argument is the atom, which is associated with the token after reduction. This atom should not necessarily be a grounded atom.
bind!
returns the unit value
()
similar to
println!
or
add-atom
.
Consider the following program:
metta
```
( = ( get-hello ) &hello ) ( = ( get-hello ) &hello ) ! ( bind! &hello ( Hello world )) ; () ! ( bind! &hello ( Hello world )) ; () ! ( get-metatype &hello ) ; Expression ! ( get-metatype &hello ) ; Expression ! &hello ; (Hello world) ! &hello ; (Hello world) ! ( get-hello ) ; &hello ! ( get-hello ) ; &hello
```
Run
We first define the function
(get-hello)
, which returns the symbol
&hello
. Then, we bind the token
&hello
to the atom
(Hello world)
. Note that the metatype of
&hello
is
Expression
, because it is replaced by the parser and gets to the interpreter already as
(Hello world)
.
! &hello
is expectedly
(Hello world)
. Once again,
&hello

is not reduced to
(Hello world)
by the interpreter. It is replaced with it by the parser. It can be seen by the fact that
(get-hello)
returns
&hello
as a symbol, because it was parsed and added to the program space before
bind!
.
bind!
might be tempting to use to refer to some lengthy constant expressions, e.g.
metta
! (bind! &x (foo1 (foo2 3) 45 (A (v)))) ! (bind! &x (foo1 (foo2 3) 45 (A (v)))) ! &x ! &x
Run
However, this lengthy expression will be inserted to the program in place of every occurrence of
&x
. However, let us note again that the second argument of
bind!
is evaluated before
bind!
is called, which is especially important with functions with side effects. For example, the following program will print
"test"
only once, while
&res
will be simply replaced with
()
.
metta
! (bind! &res (println! "test")) ! (bind! &res (println! "test")) ! &res ! &res ! &res ! &res
Run
Using
bind!
for unique grounded atoms intensively used in the program can be more reasonable. Binding spaces created with
(new-space)
to tokens is one of possible use cases:
metta
! (bind! &space (new-space)) ; () ! (bind! &space (new-space)) ; () ! (add-atom &space (Parent Bob Ann)) ; () ! (add-atom &space (Parent Bob Ann)) ; () ! &space ; GroundingSpace-addr ! &space ; GroundingSpace-addr ! (match &space (Parent $ x $ y) ($ x $ y)) ; (Bob Ann) ! (match &space (Parent $ x $ y) ($ x $ y)) ; (Bob Ann) ! (match &self (Parent $ x $ y) ($ x $ y)) ; empty ! (match &self (Parent $ x $ y) ($ x $ y)) ; empty
Run
However, if spaces are created dynamically depending on runtime data,

bind!
is not usable.
Imports

Stdlib has operations for importing scripts and modules. One such operation is
import!
. It accepts two arguments. The first argument is a symbol, which is turned into the token for accessing the imported module. The second argument is the module name. For example, the program from the
tutorial
could be split into two scripts - one containing knowledge, and another one querying it.
metta
; people_kb.metta ; people_kb.metta (Female Pam) (Female Pam) (Male Tom) (Male Tom) (Male Bob) (Male Bob) (Female Liz) (Female Liz) (Female Pat) (Female Pat) (Female Ann) (Female Ann) (Male Jim) (Male Jim) (Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) (Parent Bob Pat) (Parent Bob Pat) (Parent Pat Jim) (Parent Pat Jim)
metta
; main.metta ; main.metta ! (import! &people people_kb) ! (import! &people people_kb) (= (get-sister $ x) (= (get-sister $ x) (match &people (match &people (, (Parent $ y $ x) (, (Parent $ y $ x) (Parent $ y $ z) (Parent $ y $ z) (Female $ z)) (Female $ z)) $ z $ z)))) ! (get-sister Bob) ! (get-sister Bob)
Here,
(import! &people people_kb)
looks similar to
(bind! &people (new-space))
, but
import!
fills in the loaded space with atoms from the script. Let us note that
import!
does more work than just loading the script into a space. It interacts with the module system, which is described in another tutorial.
&self
can be passed as the first argument to
import!
. In this case, the script or module will still be loaded into a separate space, but the atom wrapping this space will be inserted to
&self
. Pattern matching queries encountering such atoms will delegate queries to them (with the exception, when the space atom itself matches against the query, which happens, when this query is just a variable, e.g.,
$x
). Thus, it works similar to inserting all the atoms to
&self
, but with some differences, when importing the same module happens multiple times, say, in different submodules.
One may use

get-atoms
method to see that the empty MeTTa script is not that empty and contains the stdlib space(s). Note that the result
get-atoms
will be reduced. Thus, it is not recommended to use in general.
metta
! (get-atoms &self) ! (get-atoms &self)
Run
Some space atoms are present in the seemingly empty program since some modules are pre-imported. Indeed, one can find, say,
if
definition in
&self
, which actually resides in the stdlib space inserted into
&self
as an atom
metta
! (match &self ! (match &self (= (if $ cond $ then $ else) $ result) (= (if $ cond $ then $ else) $ result) (quote (= (if $ cond $ then $ else) $ result)) (quote (= (if $ cond $ then $ else) $ result))))
Run
mod-space!
returns the space of the module (and tries to load the module if it is not loaded into the module system). Thus, we can explore the module space explicitly.
metta
! (mod-space! stdlib) ! (mod-space! stdlib) ! (match (mod-space! stdlib) ! (match (mod-space! stdlib) (= (if $ cond $ then $ else) $ result) (= (if $ cond $ then $ else) $ result) (quote (= (if $ cond $ then $ else) $ result)) (quote (= (if $ cond $ then $ else) $ result))))
Run
Control flow

MeTTa has several specific constructs that allow a program to execute different parts of code based either on pattern matching or logical conditions.
if

if
was already covered in
this tutorial
. But let us recap it as a part of stdlib.
The
if
statement implementation in MeTTa can be the following function
metta
(: if (-> Bool Atom Atom $ t)) (: if (-> Bool Atom Atom $ t)) (= (if True $ then $ else) $ then) (= (if True $ then $ else) $ then) (= (if False $ then $ else) $ else) (= (if False $ then $ else) $ else)
Here, the first argument (condition) is
Bool
, which is evaluated before executing the equality-query for

if
.
The next two arguments are not evaluated and returned for the further evaluation depending on whether the first argument is matched with
True
or
False
.
The basic use of
if
in MeTTa is similar to that in other languages:
metta
```
( = ( foo $ x ) ( = ( foo $ x ) ( if ( >= $ x 0 ) ( if ( >= $ x 0 ) ( + $ x 10 ) ( + $ x 10 ) ( * $ x -1 ) ( * $ x -1 ) ) ) ) ) ! ( foo 1 ) ; 11 ! ( foo 1 ) ; 11 ! ( foo -9 ) ; 9 ! ( foo -9 ) ; 9
```
Run
Here we have a function
foo
that adds
10
to the input value if it's grater or equal
0
, and multiplies the input value by
-1
otherwise. The expression
(>= $x 0)
is the first argument of the
if
function, and it is evaluated to a
Bool
value. According to that value the expression
(+ $x 10)
or
(* $x -1)
is returned for the final evaluation, and we get the result.
In contrast to other languages, one can pass a variable to
if
and it will be matched against equalities with both
True
and
False
. Consider the following example
metta
```
! ( if $ x ( + 6 1 ) ( - 7 2 )) ! ( if $ x ( + 6 1 ) ( - 7 2 )) ( = ( foo $ b $ x ) ( = ( foo $ b $ x ) ( if $ b ( if $ b ( + $ x 10 ) ( + $ x 10 ) ( * $ x -1 ) ( * $ x -1 ) ) ) ) ) ! (( foo $ b 1 ) $ b ) ; [(-1 False), (11 True)] ! (( foo $ b 1 ) $ b ) ; [(-1 False), (11 True)]
```
Run
foo
accepts the condition for

if
, and when we pass a variable, both branches are evaluated with the corresponding binding for
$b
.
if
can also remain unreduced:
metta
```
! ( if ( > $ x 0 ) ( + $ x 5 ) ( - $ x 5 )) ! ( if ( > $ x 0 ) ( + $ x 5 ) ( - $ x 5 ))
```
Run
In this expression,
(> $x 0)
remains unreduced. Its overall type is
Bool
, but it can't be directly matched against neither
True
nor
False
. Thus, no equality is applied.
let

let
has been briefly described
in another tutorial
. Here, we will recap it.
The
let
function is utilized to establish temporary variable bindings within an expression. It allows introducing variables, assign values to them, and then use these values within the scope of the
let
block.
Once the
let
block has run, these variables cease to exist and any previous bindings are re-established. Depending on the interpreter version,
let
can be either a basic grounded function, or be implemented using other primitives.
Let us consider its type
metta
```
! ( get-type let ) ! ( get-type let )
```
Run
The first argument of
let
is a pattern of
Atom
type, which is not evaluated. The second argument is the value, which is reduced before being passed to
let

The third parameter is an
Atom
again. An attempt to unify the first two arguments is performed. If it succeeds, the found bindings are substituted to the third argument, which is then evaluated. Otherwise, the empty result is returned.
Consider the following example:
metta
(= (test 1) 1) (= (test 1) 1) (= (test 1) 0) (= (test 1) 0) (= (test 2) 2) (= (test 2) 2) ! (let $ W (test $ X) (println! ("test" $ X => $ W))) ! (let $ W (test $ X) (println! ("test" $ X => $ W)))
Run
The code above will print:
metta
("test" 1 => 1) ("test" 1 => 1) ("test" 1 => 0) ("test" 1 => 0) ("test" 2 => 2) ("test" 2 => 2)
and return three unit results produced by
println!
. It can be seen that variables from both the first and the second arguments can appear in the third argument.
The following example shows the difference between the first two arguments.
metta
(= (test 1) 2) (= (test 1) 2) ! (let 2 (test 1) YES) ; YES ! (let 2 (test 1) YES) ; YES ! (let (test 1) 2 NO) ; empty ! (let (test 1) 2 NO) ; empty
Run
In case of
(let 2 (test 1) YES)
,
(test 1)
is evaluated to
2
, and it can be unified with the first argument, which is also
2
. In case of
(let (test 1) 2 NO)
,
(test 1)
is not reduced, and it cannot be unified (as a pattern) with
2
, so the overall result is empty.
This example also shows that variables are not mandatory in
let
. What is needed is the possibility to unify the arguments. This allows using
let
for chaining operations, and this chaining can be conditional if the first operation returns some value, e.g.
metta
(= (is-frog Sam) True) (= (is-frog Sam) True) (= (print-if-frog $ x) (= (print-if-frog $ x) (let True (is-frog $ x) (let True (is-frog $ x) (println! ($ x is frog!)))) (println! ($ x is frog!)))) ! (print-if-frog Sam)

; () ! (print-if-frog Sam) ; () ! (print-if-frog Ben) ; empty ! (print-if-frog Ben) ; empty
Run
Another basic use of
let
is to calculate values for passing them to functions accepting arguments of
Atom
type, for example:
metta
(Sam is 34 years old) (Sam is 34 years old) ! (match &self ($ who is (+ 20 14) years old) $ who) ; empty ! (match &self ($ who is (+ 20 14) years old) $ who) ; empty ! (let $ r (+ 20 14) ! (let $ r (+ 20 14) (match &self ($ who is $ r years old) $ who)) ; Sam (match &self ($ who is $ r years old) $ who)) ; Sam
Run
Since the first argument can be not only a variable or a concrete value, but also an expression,
let
can be used for deconstructing expressions
metta
(= (fact Sam) (age 34)) (= (fact Sam) (age 34)) (= (fact Sam) (color green)) (= (fact Sam) (color green)) (= (fact Tom) (age 14)) (= (fact Tom) (age 14)) ! (let (age $ r) (fact $ who) ! (let (age $ r) (fact $ who) ($ who is $ r)) ; [(Tom is 14), (Sam is 34)] ($ who is $ r)) ; [(Tom is 14), (Sam is 34)]
Run
The branches not corresponding to the
(age $r)
pattern are filtered out.
let*

When several consecutive substitutions are required,
let*
can be used for convenience. The first argument of
let*
is
Expression
, which elements are the required substitutions, while the second argument is the resulting expression. In the following example, several values are subsequently calculated, and
let*
allows making it more readable (notice also how pattern matching helps to calculate minimum and maximum values together with their absolute difference in one
if
).
metta
(Sam is 34) (Sam is 34) (Tom is 14) (Tom is 14) (= (person-by-age $ age) (= (person-by-age $ age) (match &self ($ who is $ age) $ who)) (match &self ($ who is $ age) $ who)) (= (persons-of-age $ a $ b) (= (persons-of-age $ a $ b) (let* ((($ age-min $ age-max $ diff) (let* ((($ age-min $ age-max $ diff

) (if (< $ a $ b) (if (< $ a $ b) ($ a $ b (- $ b $ a)) ($ a $ b (- $ b $ a)) ($ b $ a (- $ a $ b)))) ($ b $ a (- $ a $ b)))) ($ younger (person-by-age $ age-min)) ($ younger (person-by-age $ age-min)) ($ older (person-by-age $ age-max)) ($ older (person-by-age $ age-max)))) ($ younger is younger than $ older by $ diff years)) ($ younger is younger than $ older by $ diff years)))) ! (persons-of-age 34 14) ! (persons-of-age 34 14)
Run
Another case, for which
let*
can be convenient, is the consequent execution of side-effect functions, e.g.
metta
(Sam is 34) (Sam is 34) (= (age++ $ who) (= (age++ $ who) (let* (($ age (match &self ($ who is $ a) $ a)) (let* (($ age (match &self ($ who is $ a) $ a)) (() (println! (WAS: ($ who is $ age)))) (() (println! (WAS: ($ who is $ age)))) (() (remove-atom &self ($ who is $ age))) (() (remove-atom &self ($ who is $ age))) (() (add-reduct &self ($ who is (+ $ age 1)))) (() (add-reduct &self ($ who is (+ $ age 1)))) ($ upd (match &self ($ who is $ a) $ a)) ($ upd (match &self ($ who is $ a) $ a)) (() (println! (NOW: ($ who is $ upd))))) (() (println! (NOW: ($ who is $ upd))))) $ upd $ upd)))) ! (age++ Sam) ; 35 ! (age++ Sam) ; 35
Run
case

Another type of multiway control flow mechanism in MeTTa is the
case
function, which was briefly mentioned in
the tutorial
. It turns
let
around and subsequently tests multiple pattern-matching conditions for the given value. This value is provided by the first argument. While the formal argument type is
Atom
, it will be evaluated. The second argument is a tuple, which elements are pairs mapping condition patterns to results.
metta
(Sam is Frog) (Sam is Frog) (Apple is Green) (Apple is Green) (= (test $ who) (= (test $ who) (case (match &self ($ who is $ x) $ x) (case (match &self ($ who is $ x) $ x) (((42 "The answer is 42!") (42 "The answer is 42!") (Frog "Do not ask me about frogs") (Frog "Do not ask me about frogs") ($ a ($ who is $ a)) ($ a ($ who is $ a)))))))) ! (test Sam) ; "Do not ask me about frogs" ! (test Sam) ; "Do not ask me about frogs" ! (test Apple) ; (Apple is Green) ! (test Apple) ; (Apple is Green) ! (test Car) ; empty ! (test Car) ; empty
Run
Cases are processed sequentially from the first to the last. In the example above,
$a
condition will always be matched, so it is put at the end, and the corresponding branch is triggered, when all the previous conditions are not met. Note, however, that

$a
is not matched against the empty result in the last case.
In order to handle such cases, one can use
Empty
symbol as a case pattern (in some versions of the interpreter,
Empty
is the dedicated symbol which
(empty)
is evaluated to). The following code should return
"Input was really empty"
:
metta
! (case (empty) ! (case (empty) (($ _ "Should not be the case") (($ _ "Should not be the case") (Empty "Input was really empty")) (Empty "Input was really empty"))))
Run
Let us consider the use of patterns in
case
on example of the rock-paper-scissors game. There are multiple ways of how to write a function, which will return the winner. The following function uses one
case
with five branches:
sandbox
metta
(= (rps-winner $ x $ y) (= (rps-winner $ x $ y) (case ($ x $ y) (case ($ x $ y) (((Paper Rock) First) (((Paper Rock) First) ((Scissors Paper) First) ((Scissors Paper) First) ((Rock Scissors) First) ((Rock Scissors) First) (($ a $ a) Draw) (($ a $ a) Draw) ($ _ Second)) ($ _ Second)))))) ! (rps-winner Paper Scissors) ; Second ! (rps-winner Paper Scissors) ; Second ! (rps-winner Rock Scissors) ; First ! (rps-winner Rock Scissors) ; First ! (rps-winner Paper Paper) ; Draw ! (rps-winner Paper Paper) ; Draw
Run
Copied
Reset
One could also write a function, which checks if the first player wins, and use it twice (for
($x $y)
and
($y $x)
). This could be more scalable for game extensions with additional gestures, and could be more robust to unexpected inputs (although this should be better handled with types). You can try experimenting with different approaches using the sandbox above.
Operations over atoms

Stdlib contains operations to construct and deconstruct atoms as instances of
Expression
meta-type. Let us first describe these operations.
Deconstructing expressions

car-atom
and
cdr-atom
are fundamental operations that are used to manipulate atoms. They are named after 'car' and 'cdr' operations in Lisp and other similar programming languages.
The
car-atom
function extracts the first atom of an expression as a tuple.
metta
```
! ( get-type car-atom ) ; (-> Expression %Undefined%) ! ( get-type car-atom ) ; (-> Expression %Undefined%) ! ( car-atom ( 1 2 3 )) ; 1 ! ( car-atom ( 1 2 3 )) ; 1 ! ( car-atom ( Cons X Nil )) ; Cons ! ( car-atom ( Cons X Nil )) ; Cons ! ( car-atom ( seg ( point 1 1 ) ( point 1 4 ))) ; seg ! ( car-atom ( seg ( point 1 1 ) ( point 1 4 ))) ; seg
```
Run
The
cdr-atom
function extracts the tail of an expression, that is, all the atoms of the argument except the first one.
metta
```
! ( get-type cdr-atom ) ; (-> Expression %Undefined%) ! ( get-type cdr-atom ) ; (-> Expression %Undefined%) ! ( cdr-atom ( 1 2 3 )) ; (2 3) ! ( cdr-atom ( 1 2 3 )) ; (2 3) ! ( cdr-atom ( Cons X Nil )) ; (X Nil) ! ( cdr-atom ( Cons X Nil )) ; (X Nil) ! ( cdr-atom ( seg ( point 1 1 ) ( point 1 4 ))) ; ((point 1 1) (point 1 4)) ! ( cdr-atom ( seg ( point 1 1 ) ( point 1 4 ))) ; ((point 1 1) (point 1 4))
```
Run
Constructing expressions

cons-atom
is a function, which constructs an expression using two arguments, the first of which serves as a head and the second serves as a tail.
metta
```
! ( get-type cons-atom ) ; (-> Atom Expression Expression) ! ( get-type cons-atom ) ; (-> Atom Expression Expression) ! ( cons-atom 1 ( 2 3 )) ; (1 2 3) ! ( cons-atom 1 ( 2 3 )) ; (1 2 3) ! ( cons-atom Cons ( X Nil )) ; (Cons X Nil) ! ( cons-atom Cons ( X Nil )) ; (Cons X Nil) ! ( cons-atom seg (( point 1 1 ) ( point 1 4 ))) ; (seg (point 1 1) (point 1 4)) ! ( cons-atom seg (( point 1 1 ) ( point 1 4 ))) ; (seg (point 1 1) (point 1 4))
```
Run
cons-atom
reverses the results of
car-atom
and
cdr-atom
:
metta
```
( = ( reconstruct $ xs ) ( = ( reconstruct $ xs ) ( let* (( $ head ( car-atom $ xs )) ( let* (( $ head ( car-atom $ xs )) ( $ tail ( cdr-atom $ xs ))) ( $ tail ( cdr-atom $ xs ))) ( cons-atom $ head $ tail )) ( cons-atom $ head $ tail )) ) ) ! ( reconstruct ( 1 2 3 )) ; (1 2 3) ! ( reconstruct ( 1 2 3 )) ; (1 2 3) ! (
```

reconstruct (Cons X Nil)) ; (Cons X Nil) ! (reconstruct (Cons X Nil)) ; (Cons X Nil)
Run
Note that we need
let
in the code above, because
cons-atom
expects "meta-typed" arguments, which are not reduced. For example,
cdr-atom
will not be evaluated in the following code:
metta
! (cons-atom 1 (cdr-atom (1 2 3))) ; (1 cdr-atom (1 2 3)) ! (cons-atom 1 (cdr-atom (1 2 3))) ; (1 cdr-atom (1 2 3))
Run
Let us consider how basic recursive processing of expressions can be implemented:
metta
(: map-expr (-> (-> $ t $ t) Expression Expression)) (: map-expr (-> (-> $ t $ t) Expression Expression)) (= (map-expr $ f $ expr) (= (map-expr $ f $ expr) (if (== $ expr ()) () (if (== $ expr ()) () (let* (($ head (car-atom $ expr)) (let* (($ head (car-atom $ expr)) ($ tail (cdr-atom $ expr)) ($ tail (cdr-atom $ expr)) ($ head-new ($ f $ head)) ($ head-new ($ f $ head)) ($ tail-new (map-expr $ f $ tail)) ($ tail-new (map-expr $ f $ tail)))) (cons-atom $ head-new $ tail-new) (cons-atom $ head-new $ tail-new))))))) ! (map-expr not (False True False False)) ! (map-expr not (False True False False))
Run
Comparison with custom data constructors

A typical way to construct lists using custom data structures is to introduce a symbol, which can be used for pattern-matching. Then, extracting heads and tails of lists becomes straightforward, and special functions for this are not need. They can be easily implemented via pattern-matching:
metta
(= (car (Cons $ x $ xs)) $ x) (= (car (Cons $ x $ xs)) $ x) (= (cdr (Cons $ x $ xs)) $ xs) (= (cdr (Cons $ x $ xs)) $ xs) ! (cdr (Cons 1 (Cons 2 (Cons 3 Nil)))) ! (cdr (Cons 1 (Cons 2 (Cons 3 Nil))))
Run
But one can implement recursive processing without
car
and
cons
:
metta
(: map (-> (-> $ t $ t) Expression Expression)) (: map (-> (-> $ t $ t) Expression Expression)) (= (map $ f Nil) Nil) (= (map $ f Nil) Nil) (= (map $ f (Cons $ x $ xs)) (= (map $ f (Cons $ x $ xs)) (Cons ($ f $ x) (map $ f $ xs))) (Cons ($ f $ x) (map $ f $ xs))) ! (map not (Cons False (Cons True (Cons False (Cons False Nil))))) ! (map not (Cons False (Cons True (Cons False (Cons False Nil)))))
Run

Instead of
Expression
, one would typically use a polymorphic
List
type (as described another tutorial).
Implementing
map
with the use of pattern matching over list constructors is much simpler. Why can't it be made with
cons-atom
?
cons-atom
,
car-atom
,
cdr-atom
work on the very base meta-level as grounded functions. If we introduced explicit constructors for expressions, then we would just move this meta-level further, and the question would arise how expressions with these new constructors are constructed. Apparently, we need to stop somewhere and introduce the very basic operations to construct all other composite expressions. Using explicit data constructors should typically be preferred over resorting to these atom-level operations.
Typical usage

car-atom
and
cdr-atom
are typically used for recursive traversal of an expression. One basic example is creation of lists from tuples. In case of reducible non-nested lists, the code is simple:
metta
```
( = ( to-list $ expr ) ( = ( to-list $ expr ) ( if ( == $ expr ()) Nil ( if ( == $ expr ()) Nil ( Cons ( car-atom $ expr ) ( Cons ( car-atom $ expr ) ( to-list ( cdr-atom $ expr ))) ( to-list ( cdr-atom $ expr ))) ) ) ) ) ! ( to-list ( False ( True False ) False False )) ! ( to-list ( False ( True False ) False False ))
```
Run
Parsing a tuple of arbitrary length (if the use of explicit constructors is not convenient) is a good use case for operations with expressions. For example, one may try implementing
let*
by subsequently processing the tuple of variable-value pairs and applying
let
.
One more fundamental use case for analyzing expressions is implementation of custom interpretation schemes, if they go beyond the default MeTTa interpretation process and domain specific languages. A separate tutorial will be devoted to this topic.
But let us note here that combining
car-atom
and

cdr-atom
with
get-metatype
will be a typical pattern here. Here, we provide a simple example for parsing nested tuples:
metta
(= (to-tree $ expr) (= (to-tree $ expr) (case (get-metatype $ expr) (case (get-metatype $ expr) ((Expression ((Expression (if (== $ expr ()) Nil (if (== $ expr ()) Nil (Cons (to-tree (car-atom $ expr)) (Cons (to-tree (car-atom $ expr)) (to-tree (cdr-atom $ expr))) (to-tree (cdr-atom $ expr))))))) ($ _ $ expr) ($ _ $ expr))))))) ! (to-tree (False (True False) False False)) ! (to-tree (False (True False) False False))
Run
Note the difference of the result with
to-list
. The internal
(True False)
is also converted to the list now. It happens because the head of the current tuple is also passed to
to-tree
. For this to work, we need to analyze if the argument is an expression. If it is not, the value is not transformed.
Using MeTTa from Python

Table of Contents

Running MeTTa in Python

Introduction

As Python has a broad range of applications, including web development, scientific and numeric computing, and especially AI, ML, and data analysis, its combined use with MeTTa significantly expands the possibilities of building AI systems. Both ways can be of interest:
embedding Python objects into MeTTa for serving as sub-symbolic (and, in particular, neural) components within a symbolic system;
using MeTTa from Python for defining knowledge, rules, functions, and variables which can be referred to in Python programs to create prompt templates for LLMs, logical reasoning, or compositions of multiple AI agents.
We start with the use of MeTTa from Python via high-level API, and then we will proceed to a tighter integration.
Setup

Firstly, you need to have MeTTa's Python API installed as a Python package. MeTTa itself can be built from source with Python support and installed in the development mode in accordance with the instructions in the
github repository

. This approach is more involved, but it will yield the latest version with a number of configuration options.
However, for a quick start, hyperon package available via
pip
under Linux or MacOs (possibly except for newest processors):
bash
pip install hyperon pip install hyperon
MeTTa runner class

The main interface class for MeTTa in Python is
MeTTa
class, which represents a runner built on top of the interpreter to execute MeTTa programs. It can be imported from
hyperon
package and its instance can be created and used to run MeTTa code directly:
python
from hyperon import MeTTa from hyperon import MeTTa metta = MeTTa() metta = MeTTa() result = metta.run(''' result = metta.run(''' (= (foo) boo) (= (foo) boo) ! (foo) ! (foo) ! (match &self (= ($f) boo) $f) ! (match &self (= ($f) boo) $f) ''') ''') print (result) # [[boo], [foo]] print (result) # [[boo], [foo]]
Run
The result of
run
is a list of results of all evaluated expressions (following the exclamation mark
!
). Each of this results is also a list (each containing one element in the example above). These results are not printed to the console by
metta.run
. They are just returned. Thus, we print them in Python.
Let us note that
MeTTa
instance preserve their program space after
run
has finished. Thus,
run
can be executed multiple times:
python
from hyperon import MeTTa from hyperon import MeTTa metta = MeTTa() metta = MeTTa() metta.run(''' metta.run(''' (Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) ''') ''') print (metta.run('!(match &self (Parent Tom $x) $x)')) # [[Liz, Bob]] print (metta.run('!(match &self (Parent Tom $x) $x)')) # [[Liz, Bob]] print (metta.run('!(match &self (Parent $x Bob) $x)')) # [[Tom, Pam]] print (metta.run('!(match &self (Parent $x Bob) $x)')) # [[Tom, Pam]]
Run
Parsing MeTTa code

The runner has methods for parsing a program code instead of executing it. Parsing produces MeTTa atoms wrapped into Python objects (so they can be manipulated from Python). Creating a simple expression atom

(A B)
looks like
python
atom = metta.parse_single('(A B)') atom = metta.parse_single('(A B)')
The
parse_single()
method parses only the next single token from the text program, thus the following example will give equivalent results
python
from hyperon import MeTTa from hyperon import MeTTa metta = MeTTa() metta = MeTTa() atom1 = metta.parse_single('(A B)') atom1 = metta.parse_single('(A B)') atom2 = metta.parse_single('(A B) (C D)') atom2 = metta.parse_single('(A B) (C D)') print (atom1) # (A B) print (atom1) # (A B) print (atom2) # (A B) print (atom2) # (A B)
Run
The
parse_all()
method can be used to parse the whole program code given in the string and get the list of atoms
python
from hyperon import MeTTa from hyperon import MeTTa metta = MeTTa() metta = MeTTa() program = metta.parse_all('(A B) (C D)') program = metta.parse_all('(A B) (C D)') print (program) # [(A B), (C D)] print (program) # [(A B), (C D)]
Run
Accessing the program Space

Let us recall that Atomspace (or just Space) is a key component of MeTTa. It is essentially a knowledge representation database (which can be thought of as a metagraph) and the associated MeTTa functions are used for storing and manipulating information.
One can get a reference to the current program Space, which in turn may be accessed directly, wrapped in some way, or passed to the MeTTa interpreter. Having the reference, one can add new atoms into it using the
add_atom()
method
python
metta.space().add_atom(atom) metta.space().add_atom(atom)
Now let us call the
run()
method that runs the code from the program string containing a symbolic expression
python
from hyperon import MeTTa from hyperon import MeTTa metta = MeTTa() metta = MeTTa() atom = metta.parse_single('(A B)') atom = metta.parse_single('(A B)') metta.space().add_atom(atom) metta.space().add_atom(atom) print (metta.run("!(match &self (A $x) $x)")) # [[B]] print (metta.run("!(match &self (A $x) $x)")) # [[B]]
Run
The program passed to
run
contains only one expression
!(match &self (A $x) $x)

. It calls the
match
function for the pattern
(A $x)
and returns all matches for the
$x
variable. The result will be
[[B]]
, which means that
add_atom
has added
(A B)
expression extracted from the string by
parse_single
. The code
python
atom = metta.parse_single('(A B)') atom = metta.parse_single('(A B)')
metta.space().add_atom(atom) metta.space().add_atom(atom)
is effectively equivalent to
python
metta.run('(A B)') metta.run('(A B)')
because expressions are not preceded by
!
are just added to the program Space.
Please note that
python
atom = metta.parse_all('(A B)') atom = metta.parse_all('(A B)')
is not precisely equivalent to
python
metta.run('! (A B)')[0] metta.run('! (A B)')[0]
Although the results can be identical, the expression passed to
run
will be evaluated and can get reduced:
python
from hyperon import MeTTa from hyperon import MeTTa metta = MeTTa() metta = MeTTa()
print (metta.run('! (A B)')[0]) # [(A B)] print (metta.run('! (A B)')[0]) #
[(A B)] print (metta.run('! (+ 1 2)')[0]) # [3] print (metta.run('! (+ 1 2)')[
0]) # [3] print (metta.parse_all('(A B)')) # [(A B)] print (metta.parse_all('(A
B)')) # [(A B)] print (metta.parse_all('(+ 1 2)')) # [(+ 1 2)] print
(metta.parse_all('(+ 1 2)')) # [(+ 1 2)]
Run
parse_single
or
parse_all
are more useful, when we want not to add atoms to the program Space, but when we
want to get these atoms without reduction and to process them further in Python.
Besides
add_atom
(and
remove_atom

as well), Space objects have
query
method.
python
metta = MeTTa() metta = MeTTa() metta.run(''' metta.run(''' (Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) ''') ''') pattern = metta.parse_single('(Parent $x Bob)') pattern = metta.parse_single('(Parent $x Bob)') print (metta.space().query(pattern)) # [{ $x <- Pam }, { $x <- Tom }] print (metta.space().query(pattern)) # [{ $x <- Pam }, { $x <- Tom }]
Run
In contrast to
match
in MeTTa itself,
query
doesn't take the output pattern, but just returns options for variable bindings, which can be useful for further custom processing in Python. It would be useful to have a possibility to define patterns directly in Python instead of parsing them from strings.
MeTTa atoms in Python

Class
Atom
in Python (see its
implementation
) is used to wrap all atoms created in the backend of MeTTa into Python objects, so they can be manipulated in Python. An atom of any kind (metatype) can be created as an instance of this class, but classes
SymbolAtom
,
VariableAtom
,
ExpressionAtom
and
GroundedAtom
together with helper functions are inherited from
Atom
for convenience.
SymbolAtom

Symbol atoms are intended for representing both procedural and declarative knowledge entities for fully introspective processing. Such symbolic representations can be used and manipulated to infer new knowledge, make decisions, and learn from experience. It's a way of handling and processing abstract and arbitrary information.
The helper function
S()
is a convenient tool to construct an instance of
SymbolAtom
Python class. Its only specific method is

get_name
, since symbols are identified by their names. All instances of
Atom
has
get_metatype
method, which returns the atom metatype maintained by the backend.
python
from hyperon import S, SymbolAtom, Atom from hyperon import S, SymbolAtom, Atom
symbol_atom = S('MyAtom') symbol_atom = S('MyAtom') print
(symbol_atom.get_name()) # MyAtom print (symbol_atom.get_name()) # MyAtom print
(symbol_atom.get_metatype()) # AtomKind.SYMBOL print (symbol_atom.get_metatype()) #
AtomKind.SYMBOL print (type (symbol_atom)) # SymbolAtom print (type (symbol_atom))
SymbolAtom print (isinstance (symbol_atom, SymbolAtom)) # True print (isinstance
(symbol_atom, SymbolAtom)) # True print (isinstance (symbol_atom, Atom)) # True
print (isinstance (symbol_atom, Atom)) # True
Run
Let us note that
S('MyAtom')
is a direct way to construct a symbol atom without calling the parser as in
metta.parse_single('MyAtom')
. It allows constructing symbols with the use of arbitrary characters, which can be not accepted by the parser.
VariableAtom

A
VariableAtom
represents a variable (typically in an expression). It serves as a placeholder that can be matched with, or bound to other Atoms.
V()
is a convenient method to construct a
VariableAtom
:
python
from hyperon import V from hyperon import V var_atom = V('x') var_atom = V('x')
print (var_atom) # $x print (var_atom) # $x print (var_atom.get_name()) # x print
(var_atom.get_name()) # x print (var_atom.get_metatype()) # AtomKind.VARIABLE print
(var_atom.get_metatype()) # AtomKind.VARIABLE print (type (var_atom)) #
VariableAtom print (type (var_atom)) # VariableAtom
Run
VariableAtom
also has
get_name
method. Please note that variable names don't include
$
prefix in internal representation. It is used in the program code for the parser to distinguish variables and symbols.
ExpressionAtom

An
ExpressionAtom

is a list of Atoms of any kind, including expressions. It has the
get_children()
method that returns a list of all children Atoms of an expression.
E()
is a convenient method to construct expressions, it takes a list of atoms as an input. The example below shows that queries can be constructed in Python and the resulting expressions can be processed in Python as well.
python
```
from hyperon import E, S, V, MeTTa from hyperon import E, S, V, MeTTa  metta = MeTTa() metta = MeTTa() expr_atom = E(S( 'Parent' ), V( 'x' ), S( 'Bob' )) expr_atom = E(S( 'Parent' ), V( 'x' ), S( 'Bob' )) print (expr_atom) # (Parent $x Bob) print (expr_atom) # (Parent $x Bob) print (expr_atom.get_metatype()) # AtomKind.EXPR print (expr_atom.get_metatype()) # AtomKind.EXPR print (expr_atom.get_children()) # [Parent, $x, Bob] print (expr_atom.get_children()) # [Parent, $x, Bob] # Let us use expr_atom in the query # Let us use expr_atom in the query metta = MeTTa() metta = MeTTa() metta.run( ''' metta.run( ''' (Parent Tom Bob) (Parent Tom Bob) (Parent Pam Bob) (Parent Pam Bob) (Parent Tom Liz) (Parent Tom Liz) (Parent Bob Ann) (Parent Bob Ann) ''' ) ''' ) print (metta.space().query(expr_atom)) # [{ $x <- Pam }, { $x <- Tom }] print (metta.space().query(expr_atom)) # [{ $x <- Pam }, { $x <- Tom }] result = metta.run( '! (match &self (Parent $x Bob) (Retrieved $x))' )[ 0 ] result = metta.run( '! (match &self (Parent $x Bob) (Retrieved $x))' )[ 0 ] print (result) # [(Retrieved Tom) (Retrieved Pam)] print (result) # [(Retrieved Tom) (Retrieved Pam)] # Ignore 'Retrieved' in expressions and print Pam, Tom # Ignore 'Retrieved' in expressions and print Pam, Tom for r in result: for r in result: print (r.get_children()[ 1 ]) print (r.get_children()[ 1 ])
```
Run
GroundedAtom

GroundedAtom
is a special subtype of
Atom
that makes a connection between the abstract, symbolically represented knowledge within AtomSpace and the external environment or the behaviors/actions in the outside world. Grounded Atoms often have an associated piece of program code that can be executed to produce specific output or trigger an action.
For example, this could be used to pull in data from external sources into the AtomSpace, to run a PyTorch model, to control an LLM agent, or to perform any other action that the system needs to interact with the external world, or just to perform intensive computations.
Besides the content, which a
GroundedAtom
wraps, there are three other aspects which can be customized:
the type of GroundedAtom (kept within the Atom itself);
the matching algorithm used by the Atom;
a GroundedAtom can be made executable, and used to apply sub-symbolic operations to other Atoms as arguments.
Let us start with basic usage.
G()
is a convenient method to construct a
GroundedAtom

. It can accept any Python object, which has
copy
method. In the program below, we construct an expression with a custom grounded atom and add it to the program Space. Then, we perform querying to retrieve this atom.
GroundedAtom
has
get_object()
method to extract the data wrapped into the atom.
python
```
from hyperon import * from hyperon import * metta = MeTTa() metta = MeTTa() entry = E(S( 'my-key' ), G({ 'a' : 1 , 'b' : 2 })) entry = E(S( 'my-key' ), G({ 'a' : 1 , 'b' : 2 })) metta.space().add_atom(entry) metta.space().add_atom(entry) result = metta.run( '! (match &self (my-key $x) $x)' )[ 0 ][ 0 ] result = metta.run( '! (match &self (my-key $x) $x)' )[ 0 ][ 0 ] print ( type (result)) # GroundedAtom print ( type (result)) # GroundedAtom print (result.get_object()) # {'a': 1, 'b': 2} print (result.get_object()) # {'a': 1, 'b': 2}
```
Run
As the example shows, we can add a custom grounded object to the space, query and get it in MeTTa, and retrieve back to Python.
However, wrapping Python object directly to
G()
is typically not recommended. Python API for MeTTa implements a generic class
GroundedObject
with the field
content
storing a Python object of interest and the
copy
method. There are two inherited classes,
ValueObject
and
OperationObject
with some additional functionality. Methods
ValueAtom
and
OperationAtom
is a sugared way to construct
G(ValueObject(...))
and
G(OperationObject(...))
correspondingly. Thus, it would be preferable to use
ValueAtom({'a': 1, 'b': 2})
in the code above, although one would need to write
result.get_object().content
to access the corresponding Python object (
ValueObject
has a getter
value
for
content
as well, while

OperationObject
uses
op
for this).
The constructor of
GroundedObject
accepts the
content
argument (a Python object) to be wrapped into the grounded atom and optionally the
id
argument (optional) for representing the atom and optionally for comparing atoms,
when utilizing the content for this is not desirable. The
ValueObject
class adds the getter
value
for returning the content of the grounded atom.
Arguments of the
OperationObject
constructor include
name
,
op
, and
unwrap
.
name
serves as the
id
for the grounded atom,
op
(a function) defining the operation is used as the
content
of the grounded atom, and
unwrap
(a boolean, optional) indicates whether to unwrap the
GroundedAtom
content when applying the operation (see more on
uwrap
on
the next page
of this tutorial).
While there is a choice whether to use
ValueAtom
and
OperationAtom
classes for custom objects or to directly wrap them into
G
, grounded objects constructed in the MeTTa code are returned as such sugared atoms:
python
from hyperon import * from hyperon import * metta = MeTTa() metta = MeTTa() plus =

```
metta.parse_single( '+' ) plus = metta.parse_single( '+' ) print ( type
(plus.get_object())) # OperationObject print ( type (plus.get_object())) #
OperationObject print (plus.get_object().op) # some lambda print
(plus.get_object().op) # some lambda print (plus.get_object()) # + as a
representation of this operation print (plus.get_object()) # + as a representation
of this operation calc = metta.run( '! (+ 1 2)' )[ 0 ][ 0 ] calc = metta.run( '! (+
1 2)' )[ 0 ][ 0 ] print ( type (calc.get_object())) # ValueObject print ( type
(calc.get_object())) # ValueObject print (calc.get_object().value) # 3 print
(calc.get_object().value) # 3  metta.run( '(my-secret-symbol 42)' ) # add the
expression to the space metta.run( '(my-secret-symbol 42)' ) # add the expression to
the space pattern = E(V( 'x' ), ValueAtom( 42 )) pattern = E(V( 'x' ), ValueAtom( 42
)) print (metta.space().query(pattern)) # { $x <- my-secret-symbol } print
(metta.space().query(pattern)) # { $x <- my-secret-symbol }
```

Run

As can be seen from the example,
ValueAtom(42)
can be matched against
42
appeared in the MeTTa program (although it is not recommended to use grounded atoms as keys for querying).

Apparently, there is a textual representation of grounded atoms, from which atoms themselves are built by the parser. But is it possible to introduce such textual representations for custom grounded atoms, so we could refer to them in the textual program code? The answer is yes. The Python and MeTTa API for this is described on the next page.

Parsing grounded atoms

Tokenizer

The MeTTa interpreter operates with the internal representation of programs in the form of atoms. Atoms can be constructed in the course of parsing or directly using the corresponding API. Let us examine what atoms are constructed by the parser. In the following program, we parse the expression
(+ 1 S)
.

python

```
from hyperon import * from hyperon import * metta = MeTTa() metta = MeTTa() expr1 =
metta.parse_single( '(+ 1 S)' ) expr1 = metta.parse_single( '(+ 1 S)' ) expr2 = E(S(
'+' ), S( '1' ), S( 'S' )) expr2 = E(S( '+' ), S( '1' ), S( 'S' )) print ( 'Expr1: '
, expr1) print ( 'Expr1: ' , expr1) print ( 'Expr2: ' , expr2) print ( 'Expr2: ' ,
expr2) print ( 'Equal: ' , expr1 == expr2) print ( 'Equal: ' , expr1 == expr2) for
atom in expr1.get_children(): for atom in expr1.get_children(): print ( f 'type( {
atom } )= {type (atom) } ' ) print ( f 'type( { atom } )= {type (atom) } ' )
```

Run

The result of parsing differs from the expression
(+ 1 S)
composed of symbolic atoms. Indeed, the atoms constructed from
+
and
1

by the parser are grounded atoms - not symbols. At the same time,
S('+')
is already a symbol atom.
Transformation of the textual representation to grounded atoms is not hard-coded. It is done by the tokenizer on the base of a mapping from tokens in the form of regular expressions to constructors of corresponding grounded atoms.
The initial mapping is provided by the
stdlib
module, but it can be modified later. In the simple case, tokens are just strings.
For example, the tokenizer is informed that if
+
is encountered in the course of parsing, the following atom should be constructed
python
```
OperationAtom( '+' , lambda a, b: a + b, OperationAtom( '+' , lambda a, b: a + b, [
'Number' , 'Number' , 'Number' ]) [ 'Number' , 'Number' , 'Number' ])
```
Here,
['Number', 'Number', 'Number']
is a sugared way to defined the type
(-> Number Number Numer)
, which should also be represented as an atom.
Regular expressions are needed for such cases as parsing numbers. For example, integers are constructed on the base of the token
r"[-+]?\d+"
, and the constructor needs to get the token itself, so the atom is created by the following function once the token is encountered
python
```
lambda token: ValueAtom( int (token), 'Number' ) lambda token: ValueAtom( int
(token), 'Number' )
```
interpret

Once atoms are created, the interpreter doesn't rely on the tokenizer.
hyperon
module includes
interpret
, which is the function accepting the space as the context for interpretation and the atom to interpret.
python
```
from hyperon import * from hyperon import * metta = MeTTa() metta = MeTTa() expr1 =
metta.parse_single( '(+ 1 2)' ) expr1 = metta.parse_single( '(+ 1 2)' ) print
(interpret(metta.space(), expr1)) print (interpret(metta.space(), expr1)) expr2 =
E(OperationAtom( '+' , lambda a, b: a + b), expr2 = E(OperationAtom( '+' , lambda a,
b: a + b), ValueAtom( 1 ), ValueAtom( 2 )) ValueAtom( 1 ), ValueAtom( 2 )) print
(interpret(metta.space(), expr2)) print (interpret(metta.space(), expr2))
```
Run
The example above shows that the parsed expression is interpreted in the same ways as the expression atom constructed directly.
MeTTa.run
simply parses the program code expression-by-expression and puts the resulting atoms in the program space or immediately interprets them when
!

precedes the expression. Note that we could get the operation atom for
+
(which would be correctly typed) via
metta.parse_single('+')
Creating new tokens

Access to the tokenizer is provided by the
tokenizer()
method of the
MeTTa
class. However, it may not be used directly.
MeTTa
class has the
register_token
method, which is intended for registering a new token. It accepts a regular expression and a function, which will be called to construct an atom each time the token is encountered. The constructed atom should not necessarily be a grounded atom, although it is the most typical case.
If the token is a mere string, and creation of different atoms depending on a regular expression is not supposed,
register_atom
can be used. It accepts a regular expression and an atom, and calls
register_token
with the given token and with the lambda simply returning the given atom.
The following example illustrates creation of an Atomspace and wrapping it into a
GroundedAtom
python
```
from hyperon import * from hyperon import *  metta = MeTTa() metta = MeTTa()  # Getting a reference to a native GroundingSpace, # Getting a reference to a native GroundingSpace, # implemented by the MeTTa core library. # implemented by the MeTTa core library. grounding_space = GroundingSpaceRef() grounding_space = GroundingSpaceRef() grounding_space.add_atom(E(S( "A" ), S( "B" ))) grounding_space.add_atom(E(S( "A" ), S( "B" ))) space_atom = G(grounding_space) space_atom = G(grounding_space)  # Registering a new custom token based on a regular expression. # Registering a new custom token based on a regular expression. # The new token can be used in a MeTTa program. # The new token can be used in a MeTTa program. metta.register_atom( "&space" , space_atom) metta.register_atom( "&space" , space_atom) print (metta.run( "! (match &space (A $x) $x)" )) print (metta.run( "! (match &space (A $x) $x)" ))
```
Run
Parsing and interpretation

Although the interpreter works with the representation of programs in the form of atoms (as was mentioned above), and expressions should be parsed before being interpreted, the tokenizer can be changed in the course of MeTTa script execution. It is essential for the MeTTa module system (described in more detail in another tutorial).
import!
is not only loads a module code into a space. It can also modify the tokenizer with tokens declared in the module. This is the reason why a MeTTa is not first entirely

converted to atoms and then interpreted, but parsing and interpretation are intervened. Another approach would be to load all the atoms as symbols and resolve them at runtime, so the interpreter would verify if some symbols are grounded in subsymbolic data. This approach would have its benefits, and it might be chosen in the future versions of MeTTa. However, it would imply that introduction of new groundings to symbols has retrospective effect on the previous code.
We have also encountered creation of new tokens inside MeTTa programs with the use of
bind!
showing that token bindings don't have backward effect. The same is definitely true, when we create tokens using Python API:
python
```
from hyperon import * from hyperon import *  # A function to be registered # A
function to be registered def dup_str (s, n): def dup_str (s, n): r = "" r = "" for
i in range (n): for i in range (n): r += s r += s return r return r  metta = MeTTa()
metta = MeTTa() # Create an atom. "dup-str" is its internal name # Create an atom.
"dup-str" is its internal name dup_str_atom = OperationAtom( "dup-str" , dup_str)
dup_str_atom = OperationAtom( "dup-str" , dup_str)  # Interpreter will call this
operation atom provided directly # Interpreter will call this operation atom
provided directly print (interpret(metta.space(), print (interpret(metta.space(),
E(dup_str_atom, ValueAtom( "-hello-" ), ValueAtom( 3 )))) E(dup_str_atom, ValueAtom(
"-hello-" ), ValueAtom( 3 ))))  # Let us add a function calling `dup-str` # Let us
add a function calling `dup-str` metta.run( ''' metta.run( ''' (= (test-dup-str)
(dup-str "a" 2)) (= (test-dup-str) (dup-str "a" 2)) ''' ) ''' )  # The parser
doesn't know it, so dup-str will not be reduced # The parser doesn't know it, so
dup-str will not be reduced print (metta.run( ''' print (metta.run( ''' ! (dup-str
"-hello-" 3) ! (dup-str "-hello-" 3) ! (test-dup-str) ! (test-dup-str) ''' )) ''' ))
 # Now the token is registered. New expression will be reduced. # Now the token is
registered. New expression will be reduced. # However, `(= (test-dup-str) (dup-str
"a" 2))` was added # However, `(= (test-dup-str) (dup-str "a" 2))` was added #
before `dup-str` token was introduced. Thus, it will still # before `dup-str` token
was introduced. Thus, it will still # remain not reduced. # remain not reduced.
metta.register_atom( "dup-str" , dup_str_atom) metta.register_atom( "dup-str" ,
dup_str_atom) print (metta.run( ''' print (metta.run( ''' ! (dup-str "-hello-" 3) !
(dup-str "-hello-" 3) ! (test-dup-str) ! (test-dup-str) ''' )) ''' ))
```
Run
Kwargs for OperationAtom

Python supports variable number of arguments in functions. Such functions can be wrapped into grounded atoms as well.
python
```
from hyperon import * from hyperon import * def print_all ( * args): def print_all (
* args): for a in args: for a in args: print (a) print (a) return [Atoms. UNIT ]
return [Atoms. UNIT ] metta = MeTTa() metta = MeTTa() metta.register_atom(
"print-all" , OperationAtom( "print-all" , print_all)) metta.register_atom(
"print-all" , OperationAtom( "print-all" , print_all)) metta.run( '(print-all
"Hello" (+ 40 2) "World")' ) metta.run( '(print-all "Hello" (+ 40 2) "World")' )
```
Run
In cases when the function representing the operation has optional arguments with default values, the

Kwargs
keyword can be used to pass the keyword parameters. For example, let us define a grounded function
find-pos
which receives two strings and searches for the position of the second string in the first one. Let the default value for the second string be
"a"
. Additionally, this function has the third parameter which specifies whether the search should start from the left or the right, with the default value being
left=True
.
python
```
from hyperon import * from hyperon import * def find_pos (x: str , y = "a" , left = True ): def find_pos (x: str , y = "a" , left = True ): if left: if left: return x.find(y) return x.find(y) pos = x[ - 1 :].find(y) pos = x[ - 1 :].find(y) return len (x) - 1 - pos if pos >= 0 else pos return len (x) - 1 - pos if pos >= 0 else pos metta = MeTTa() metta = MeTTa() metta.register_atom( "find-pos" , OperationAtom( "find-pos" , find_pos)) metta.register_atom( "find-pos" , OperationAtom( "find-pos" , find_pos)) print (metta.run( ''' print (metta.run( ''' ! (find-pos "alpha") ; 0 ! (find-pos "alpha") ; 0 ! (find-pos (Kwargs (x "alpha") (left False))) ; 4 ! (find-pos (Kwargs (x "alpha") (left False))) ; 4 ! (find-pos (Kwargs (x "alpha") (y "c") (left False))) ; -1 ! (find-pos (Kwargs (x "alpha") (y "c") (left False))) ; -1 ''' )) ''' ))
```
Run
Hence, to set argument values using Kwargs, one needs to pass pairs of argument names and values.
Unwrapping Python objects from atoms

Above, we have introduced a summation operation as
OperationAtom('+', lambda a, b: a + b)
,where
a
and
b
are Python numbers instead of atoms.
a + b
is also not an atom. Creating of operation atoms getting Python objects is convenient, because it eliminates the necessity to retrieve values from grounded atoms and wrap the result of the operation back to the grounded atom. However, sometimes it is needed to write functions that operate with atoms themselves, and these atoms may not be grounded atoms wrapping Python objects.
Unwrapping Python values from input atoms and wrapping the result back into a grounded atom is the default behavior of
OperationAtom
, which is controlled by the parameter
unwrap
. Let us consider an example of implementing
+
while setting this parameter to
False

.
python
def plus (atom1, atom2): def plus (atom1, atom2): from hyperon import ValueAtom from hyperon import ValueAtom sum = atom1.get_object().value + atom2.get_object().value sum = atom1.get_object().value + atom2.get_object().value return [ValueAtom(sum , 'Number')] return [ValueAtom(sum , 'Number')] from hyperon import OperationAtom, MeTTa from hyperon import OperationAtom, MeTTa plus_atom = OperationAtom("plus" , plus, plus_atom = OperationAtom("plus" , plus, ['Number' , 'Number' , 'Number'], unwrap = False) ['Number' , 'Number' , 'Number'], unwrap = False) metta = MeTTa() metta = MeTTa() metta.register_atom("plus" , plus_atom) metta.register_atom("plus" , plus_atom) print (metta.run('! (plus 3 5)')) print (metta.run('! (plus 3 5)'))
Run
When
unwrap
is
False
, a function should be aware of the
hyperon
module, which can be inconvenient for purely Python functions. Thus, this setting is desirable for functions processing or creating atoms themselves. For example,
bind!
takes an atom to be bound to a token.
parse
takes a string and return an atom of any metatype constructed by parsing this string. One can imagine different custom operations, which accept and return atoms. Say, if a crossover operation in genetic algorithms would be implemented as a grounded operation, it would accept two atoms (typically, expressions), traverse them to find crossover points, and construct a child expression.
Embedding Python objects into MeTTa

py-atom

Introducing tokens for grounded atoms allows for both convenient syntax and direct representation of expressions with corresponding grounded atoms in a Space. However, wrapping all functions of rich Python libraries can be not always desirable. There is a way to invoke Python objects such as functions, classes, methods or other statements from MeTTa without additional Python code wrapping these objects into atoms.
py-atom
allows obtaining a grounded atom for a Python object imported from a given module or submodule. Let us consider usage of
numpy
as an example, which should be installed. For instance, the absolute value of a number in MeTTa can be calculated by employing the
absolute
function from the
numpy
library:
metta

! ((py-atom numpy.absolute) -5) ; 5 ! ((py-atom numpy.absolute) -5) ; 5
Run
Here,
py-atom
imports
numpy
library and returns an atom associated with the
numpy.absolute
function.
It is possible to designate types for the grounded atom in
py-atom
. For convenience, one can associate the result of
py-atom
with a token using
bind!
:
metta
! (bind! abs (py-atom numpy.absolute (-> Number Number))) ! (bind! abs (py-atom numpy.absolute (-> Number Number))) ! (+ (abs -5) 10) ; 15 ! (+ (abs -5) 10) ; 15
Run
We specify here that the constructed grounded operation can accept an argument of type
Number
and its result will be of
Number
type.
When
(abs -5)
is executed, it triggers a call to
absolute(-5)
. It can be seen that the results of executing Python objects imported via
py-atom
can then be directly utilized in other MeTTa expressions.
py-atom
can actually execute some Python code, which shouldn't be a statement like
x = 42
, but should be an expression, which evaluation produces a Python object. In the following example,
(py-atom "[1, 2, 3]")
produces a Python list, which then passed to
numpy.array
.
metta
! (bind! np-array (py-atom numpy.array)) ! (bind! np-array (py-atom numpy.array)) ! (np-array (py-atom "[1, 2, 3]")) ; array([1, 2, 3]) ! (np-array (py-atom "[1, 2, 3]")) ; array([1, 2, 3])
Run
py-atom
can be applied to functions accepting keyword arguments. Constructed grounded atoms

will also support
Kwargs
(
mentioned earlier
), which allows for passing only the required arguments to the function while skipping arguments with default values. For example, there is
numpy.arange
in NumPy, which returns evenly spaced values within a given interval.
numpy.arange
can be called with a varying number of positional arguments:
metta
! (bind! np-arange (py-atom numpy.arange)) ; () ! (bind! np-arange (py-atom numpy.arange)) ; () ! (np-arange 4) ; array([0, 1, 2, 3]) ! (np-arange 4) ; array([0, 1, 2, 3]) ! (np-arange (Kwargs (step 2) (stop 8))) ; array([0, 2, 4, 6]) ! (np-arange (Kwargs (step 2) (stop 8))) ; array([0, 2, 4, 6]) ! (np-arange (Kwargs (start 2) (stop 10) (step 3))) ; array([2, 5, 8]) ! (np-arange (Kwargs (start 2) (stop 10) (step 3))) ; array([2, 5, 8])
Run
py-dot

What if we wish to call functions from a submodule, say
numpy.random
? Accessing these functions via something like
(py-atom numpy.random.randint)
will work. However, it would be more efficient to get
numpy.random
itself as a Python object and access other objects in it.
py-dot
is introduced to carry out this operation.
metta
! (bind! np-rnd (py-atom numpy.random)) ! (bind! np-rnd (py-atom numpy.random)) ! ((py-dot np-rnd randint) 25) ! ((py-dot np-rnd randint) 25)
Run
In this case
py-dot
operates with two arguments: it takes the first argument, which is the grounded atom wrapping a Python object, and then searches for the value of an attribute within that object based on the name provided in the second argument.
This second argument can also contain objects in submodules. In the following example, we wrap
numpy
in the grounded atom:
metta
! (bind! np (py-atom numpy)) ! (bind! np (py-atom numpy)) ! ((py-dot np abs) -5) ! ((py-dot np abs) -5) ! ((py-dot np random.randint) -25 0) ! ((py-dot np random.randint) -25 0) ! ((py-dot np abs) ((py-dot np random.randint) -25 0)) ! ((py-dot np abs) ((py-dot np random.randint) -25 0))
Run
Here, when
(py-dot np random.randint)

is executed, it takes
numpy
object and searches for
random
in it and then for
randint
in
random
. The overall result is the grounded operation wrapping
numpy.random.randint
, which is then applied to some argument. Similar to
py-atom
,
py-dot
also permits the designation of types for the function, and supports
Kwargs
for arguments specification.
Binding
np
to
(py-atom numpy)
and accessing functions in it via
(py-dot np abs)
looks not more convenient than just using
(py-atom numpy.abs)
, but is slightly more efficient if
numpy.abs
is accessed multiple times.
py-dot
works for any Python object - not only modules:
metta
! ((py-dot "Hello World" swapcase)) ; "hELLO wORLD" ! ((py-dot "Hello World" swapcase)) ; "hELLO wORLD"
Run
Notice the additional brackets to call
swapcase
. The equivalent Python code is
"Hello World".swapcase()
, which also contains
()
. One more pair of brackets in MeTTa is needed, because
py-dot
is also a function.
Let us consider another example.
metta
! ((py-dot (py-atom "{5: \' f \' , 6: \' b \' }") get) 5) ! ((py-dot (py-atom "{5: \' f \' , 6: \' b \' }") get) 5)
Run
Here, a dictionary
{5: 'f', 6: 'b'}

is created by
py-atom
, and then the value corresponding to the key
5
is retrieved from this dictionary using
get
accessed via
py-dot
.
py-list
,
py-tuple
,
py-dict

While it is possible to create Python lists and dictionaries using code evaluation by
py-atom
, it can be desirable to construct these data structures by combining atoms in MeTTa.
In this context, since passing dictionaries, lists or tuples as arguments to functions in Python is very common, such dedicated functions as
py-dict
,
py-list
and
py-tuple
were introduced.
metta
! ((py-atom max) (py-list (-5 5 -3 10 8))) ; 10 ! ((py-atom max) (py-list (-5 5 -3 10 8))) ; 10 ! ((py-atom numpy.inner) ! ((py-atom numpy.inner) (py-list (1 2)) (py-list (3 4))) ; 1 * 3 + 2 * 4 = 11 (py-list (1 2)) (py-list (3 4))) ; 1 * 3 + 2 * 4 = 11
Run
In this example,
py-list
generates three Python lists:
[-5, 5, -3, 10, 8]
,
[1,2]
and
[3,4]
, which are passed to
max
and
numpy.inner
.
Of course, one can use
py-dict
,

py-list
, and
py-tuple
independently - not just as function arguments:
metta
! (py-dict (("a" "b") ("b" "c"))) ; creates a dict {"a":"b", "b":"c"} ! (py-dict (("a" "b") ("b" "c"))) ; creates a dict {"a":"b", "b":"c"} ! (py-tuple (1 5)) ; creates a tuple (1, 5) ! (py-tuple (1 5)) ; creates a tuple (1, 5) ! (py-list (1 (2 (3 "3")))) ; creates a nested list [1, [2, [3, '3']]] ! (py-list (1 (2 (3 "3")))) ; creates a nested list [1, [2, [3, '3']]]
Run
MeTTa-Motto

MeTTa-Motto
is a library that allows combining the capabilities of LLMs (Large Language Models) and MeTTa. MeTTa-Motto allows calling LLMs from MeTTa scripts, which enables prompt composition and chaining of calls to LLMs in MeTTa based on symbolic knowledge and reasoning.
Simple queries to LLMs

To make simple queries to an LLM using the MeTTa-Motto library, the following commands can be used:
metta
! (import! &self motto) ! (import! &self motto) ! (llm (Agent (chat-gpt)) ! (llm (Agent (chat-gpt)) (user "What is a black hole?")) (user "What is a black hole?"))
Here,
chat-gpt
is the name of the LLM being used. Currently, MeTTa-Motto supports the following LLMs:
ChatGPT (by OpenAI)
Claude (by Anthropic)
but more LLMs can be added if needed, and one can also use other LLMs via Langchain integration (see
below
).
Additionally, it is possible to specify the version of Chat-GPT or Claude, such as
metta
(Agent (chat-gpt "gpt-3.5-turbo")) (Agent (chat-gpt "gpt-3.5-turbo")) (Agent (anthropic-agent "claude-3-opus-20240229")) (Agent (anthropic-agent "claude-3-opus-20240229"))
In the example above, the agent may not be specified:
!(llm (user "What is a black hole?"))
. In this case, the default agent (currently,
chat-gpt
) will be used. The messages which we send to agents as parameters have the form
(ROLE "Text of the Message")
. There are 3 roles for messages:
user
,

assistant
and
system
.
llm
is a method defined in MeTTa-Motto, which passes messages to the specified agent and returns their results to MeTTa.
As a demonstration, instead of calling LLM agents, we will use the
EchoAgent
. This agent returns the message sent to it, including the role on whose behalf the message was sent
metta
! (import! &self motto) ; () ! (import! &self motto) ; () ! (llm (Agent EchoAgent) ! (llm (Agent EchoAgent) (user "The agent will return this text along with a role: user")) (user "The agent will return this text along with a role: user")) ; "user The agent will return this text along with a role: user" ; "user The agent will return this text along with a role: user"
Run
MeTTa agents

Also, as an Agent, we can specify the path to a file with a MeTTa script, which typically has a
.msa
(Metta Script Agent) extension. This script can contain any commands (expressions) in MeTTa, and may not necessarily include queries to LLMs in it, but it is supposed to run in a certain context.
For example, let us assume, there is a file named
some_agent.msa
containing the following code:
metta
! (Response ! (Response (if (== (messages) (user "Hello world.")) (if (== (messages) (user "Hello world.")) "Hi there" "Hi there" "Bye")) "Bye"))
Response
is used to indicate that this is the output of the agent. The
some_agent.msa
can be used in another script in the following manner:
metta
! (llm (Agent some_agent.msa) ! (llm (Agent some_agent.msa) (user "Hello world.")) ; Hi there (user "Hello world.")) ; Hi there
For a MeTTa agent, the new atom
(= (messages) (user "Hello world."))
will be added to the MeTTa space, in which this agent will be loaded
some_agent.msa
, allowing
(messages)
to be used within
some_agent.msa
. Usually,
.msa
agents are more complex and make use of LLM responses during processing.

Functional calls

Suppose we have a function that returns the current weather for a location passed as a parameter to this function. We want to ask about the weather in natural language, e.g. "What is the weather in New York today?", and receive information about the weather in conversational format. For such cases one can describe functions and have the LLM model intelligently select and output a JSON object containing the arguments needed to call one or more functions.
The latest OpenAI and Anthropic models have been trained to both detect when a function should to be called (depending on the input) and to respond with JSON that adheres to the function signature more closely. We can describe such functions in MeTTa-Motto too. For example, for the
get_current_weather
function, we should first describe it within
doc
section and define the function behavior:
metta
```
! ( import! &self motto ) ! ( import! &self motto ) ( = ( doc get_current_weather ) ( = ( doc get_current_weather ) ( Doc ( Doc ( description "Get the current weather for the city" ) ( description "Get the current weather for the city" ) ( parameters ( parameters ( location "the city: " ( "Tokyo" "New York" "London" )) ( location "the city: " ( "Tokyo" "New York" "London" )) ) ) ) ) ) ) ( = ( get_current_weather ( $ arg ) $ msgs ) ( = ( get_current_weather ( $ arg ) $ msgs ) ( if ( contains-str $ arg "Tokyo" ) ( if ( contains-str $ arg "Tokyo" ) "The temperature in Tokyo is 75 Fahrenheit" "The temperature in Tokyo is 75 Fahrenheit" ( if ( contains-str $ arg "New York" ) ( if ( contains-str $ arg "New York" ) "The temperature in New York is 80 Fahrenheit" "The temperature in New York is 80 Fahrenheit" ( concat-str ( concat-str "The temperature in " $ arg ) ( concat-str ( concat-str "The temperature in " $ arg ) " is 70 Fahrenheit" ) " is 70 Fahrenheit" ) ) ) ) ) ) ) ! ( llm ( Agent EchoAgent ) ! ( llm ( Agent EchoAgent ) ( user "Get the current weather for the city: London" ) ( user "Get the current weather for the city: London" )  ( function get_current_weather ) ; The temperature in London is 70 Fahrenheit. ( function get_current_weather ) ; The temperature in London is 70 Fahrenheit. ) )
```
Run
The parameters section describes the arguments of the function that should be retrieved from the user's message. The parameters can have the following properties:
name
,
type
,
description
and an enum with possible values.
The
type
property has a specific purpose. It can be provided in the form
((: parameter Atom) "Parameter description")
indicating that this
parameter
should be converted from the Python string to a MeTTa expression before passing to the function.

In our example,
concat-str
(concatenates two strings) and
contains-str
(which checks if the first string contains the second string) are grounded functions defined in MeTTa-Motto.
EchoAgent
is used for the demo purpose, but in real applications it will be any agent that supports functional calls. When a functional call is used with
EchoAgent
, arguments can be extracted from the user's message only if the message includes the function description and the parameter description concatenated with a possible value of the parameter (for example: "the city: " + London). This example is useful only for testing and demonstration purposes.
Scripts

It is convenient to store lengthy prompts and their templates for LLMs in separate files. For this reason, one can specify the path to such a file as a parameter of the
llm
method. While these files are also MeTTa files and can contain arbitrary computations, they are evaluated in a different context and are recommended to have
.mps
(MeTTa Prompt Script) extension. Basically, each such file is loaded as a MeTTa script to a space, which should contain expressions reduced to the parameters of the
llm
method. For example,
some_template.mps
file containing:
metta
(Agent (chat-gpt)) (Agent (chat-gpt)) (system ("Answer the user's questions if he asks about art, music, books, for other cases say: I can't answer your question")) (system ("Answer the user's questions if he asks about art, music, books, for other cases say: I can't answer your question"))
can be utilized from another
.metta
file:
metta
! (import! &self motto) ! (import! &self motto) ! (llm (Script some_template.mps) ! (llm (Script some_template.mps) (user "What is the name of Claude Monet's most famous painting?")) (user "What is the name of Claude Monet's most famous painting?")) ! (llm (Script some_template.mps) ! (llm (Script some_template.mps) (user "Which city is the capital of the USA?")) (user "Which city is the capital of the USA?"))
The following result will be obtained:
metta
"Claude Monet's most famous painting is called " Impression, Sunrise. "” "Claude Monet's most famous painting is called " Impression, Sunrise. "” " I can 't answer your question." " I can 't answer your question."
Notice that the parameters specified in the mps-file are combined with the

parameters specified directly. This allows separating reusable parts of prompts and utilizing them in different contexts in a composable way. In particular, if one supposes to try different LLMs with the same prompts,
Agent
should not be mentioned in the
.mps
file, but should be put to the
llm
call.
Since prompt templates are just spaces treated as parameters to
llm
, they can be created and filled in directly, but this is rarely needed.
metta
! (import! &self motto) ! (import! &self motto) ! (bind! &space (new-space)) ! (bind! &space (new-space)) ! (add-atom &space (Agent EchoAgent)) ! (add-atom &space (Agent EchoAgent)) ! (add-atom &space (user "Table")) ! (add-atom &space (user "Table")) ! (add-atom &space (user "Window")) ! (add-atom &space (user "Window")) ! (llm (Script &space)) ; "user Table user Window" ! (llm (Script &space)) ; "user Table user Window"
Run
We are using
EchoAgent
here. The result will be a concatenation of all the provided messages.
metta-chat

To store dialogue history during interaction with LLMs, we include a special dialogue agent. Let us consider an example. We will use the
metta-chat
agent for this purpose. Since the agent will be used multiple times, let us create a binding for it:
metta
! (bind! &chat (Agent (metta-chat dialog.msa))) ! (bind! &chat (Agent (metta-chat dialog.msa)))
The
metta-chat
agent stores the dialogue history in a special array named
history
. With each new message, the
history
is updated. The
history
array can be accessed from the MeTTa script (in our example, from
dialog.msa
) via the
(history)
function.
The file
dialog.msa
contains the following lines.
metta

(= (context) (= (context) (system "You are an AI assistant. (system "You are an AI assistant. Please, respond correspondingly.")) Please, respond correspondingly.")) ! (Response ! (Response (llm (Messages (context) (history) (messages)))) (llm (Messages (context) (history) (messages))))
And the dialogue can be executed:
metta
! (llm &chat (user "Hello! My name is John.")) ! (llm &chat (user "Hello! My name is John.")) ! (llm &chat (user "What do you know about the Big Bang Theory?")) ! (llm &chat (user "What do you know about the Big Bang Theory?")) ! (llm &chat (user "Do you know me name?")) ; "Yes, you mentioned earlier that your name is John. Is there anything specific you would like to know or discuss?" ! (llm &chat (user "Do you know me name?")) ; "Yes, you mentioned earlier that your name is John. Is there anything specific you would like to know or discuss?"
After the execution of the following line:
metta
! (llm (Agent (metta-chat dialog.msa)) (user "Hello ")) ! (llm (Agent (metta-chat dialog.msa)) (user "Hello "))
the new atom
metta
= (history) (Messages (user "Hello!") (assistant "Greetings, Frodo Baggins! It is a pleasure to see you. How may I be of assistance to you today?")) = (history) (Messages (user "Hello!") (assistant "Greetings, Frodo Baggins! It is a pleasure to see you. How may I be of assistance to you today?"))
will be added to the MeTTa space, created to execute
dialog.msa
.
Retrieval Agent

Sometimes we may require to pass information from various documents as parts of prompts for LLMs. However, these documents may also contain irrelevant data not pertinent to our objectives. In such cases, we can use a retrieval agent. When defining this agent, it is necessary to specify the document location or a path to one document, the chunk length for embedding creation, the desired number of chunks for the agent to return, and a designated path for storing the dataset:
metta
! (bind! &retrieval ! (bind! &retrieval (Agent (retrieval-agent "text_for_retrieval.txt" 200 2 "data"))) (Agent (retrieval-agent "text_for_retrieval.txt" 200 2 "data")))
Here the chunks size is equal to 200, and number of the closest chunks to return is 2. This agent computes embeddings for the provided texts and stores them in a dedicated database. In our case, we use ChromaDB, an open-source vector database. When the retrieval agent is invoked for a particular sentence, it first generates embeddings for the sentence and subsequently returns the closest chunks based on cosine distance metrics. For example, the text contains information about a scientist named John, then we can ask:
metta
(llm &retrieval (user "What is John working on?")) (llm &retrieval (user "What is John working on?"))
Here is the usage example of a retrieval agent with ChatGPT:
metta

```
! ( let $ question "What is John working on?" ! ( let $ question "What is John
working on?" ( llm ( Agent ( chat-gpt "gpt-3.5-turbo-0613" )) ( llm ( Agent (
chat-gpt "gpt-3.5-turbo-0613" )) ( Messages ( Messages ( system ( system ( "Taking
this information into account, answer the user question" ( "Taking this information
into account, answer the user question" ( llm &retrieval ( user $ question ))) ( llm
&retrieval ( user $ question ))) ) ) ( user $ question ) ( user $ question ) ) ) ) )
) )
```

LangChain Agents

As mentioned in this
tutorial
, by using
py-atom
and
py-dot
, you can invoke from MeTTa such Python objects as functions, classes, methods, or other statements. Taking this possibility into account, we have created agents in MeTTa-Motto that allow using LangChain components.
LangChain
is a Python framework for developing applications powered by large language models (LLMs). LangChain supports many different language models. For example, the following code uses GPT to translate text from English to French:
python

```
from langchain_openai import ChatOpenAI from langchain_openai import ChatOpenAI  llm
= ChatOpenAI( model = "gpt-3.5-turbo-0125" , temperature = 0 ) llm = ChatOpenAI(
model = "gpt-3.5-turbo-0125" , temperature = 0 ) messages = [ messages = [ (
"system" , "You are a helpful assistant that translates English to French." ), (
"system" , "You are a helpful assistant that translates English to French." ), (
"human" , "Translate this sentence from English to French: I love programming." ), (
"human" , "Translate this sentence from English to French: I love programming." ), ]
] llm.invoke(messages) llm.invoke(messages)
```

The implementation of the code, provided above, in MeTTa-Motto looks like this:
metta

```
! ( import! &self motto ) ! ( import! &self motto ) ( bind! &chat ( bind! &chat (
langchain-agent ( langchain-agent (( py-atom langchain_openai.ChatOpenAI ) ((
py-atom langchain_openai.ChatOpenAI ) ( Kwargs ( model "gpt-3.5-turbo-0125" ) (
temperature 0 ))) ( Kwargs ( model "gpt-3.5-turbo-0125" ) ( temperature 0 )))
motto/langchain_agents/langchain_agent.msa ))
motto/langchain_agents/langchain_agent.msa )) ! ( llm ( Agent &chat ) ! ( llm (
Agent &chat ) ( system "You are a helpful assistant that translates English to
French." ) ( system "You are a helpful assistant that translates English to French."
) ( user "Translate this sentence from English to French: I love programming." )) (
user "Translate this sentence from English to French: I love programming." ))
```

The grounded function
langchain-agent
has two parameters. The first is a chat model (in this case,
langchain_openai.ChatOpenAI
), which should be an instance of LangChain "Runnables" with an
invoke
method. The second parameter is the path to the file used to call the

invoke
method for the given chat model.
File
motto/langchain_agents/langchain_agent.msa
is a part of MeTTa-Motto library and contains the following lines:
metta
(py-dot ((py-dot (langchain-model) invoke) &list) content) (py-dot ((py-dot (langchain-model) invoke) &list) content)
The
&list
is used to store the entire message history. The grounded atom
langchain-model
is automatically initialized with the chat model passed to
langchain-agent
:
metta
(= (langchain-model) (= (langchain-model) ((py-atom langchain_openai.ChatOpenAI) ((py-atom langchain_openai.ChatOpenAI) (Kwargs (model "gpt-3.5-turbo-0125") (temperature 0))) (Kwargs (model "gpt-3.5-turbo-0125") (temperature 0)))) ,) ,
The
langchain-agent
can be used in the same situations as the
chat-gpt
or
metta-chat
agents.
These examples add not too much for what can be done without Langchain agents. However, if one wants to use LLMs not directly supported by MeTTa-Motto or to use some other components of Langchain together with knowledge representation and symbolic processing capabilities provided by MeTTa, then calling Langchain functions from MeTTa can be very useful. LangChain offers a variety of useful tools. These tools serve as interfaces that an agent, chain, or LLM can use to interact with the world. We can use these tools directly from MeTTa. For example, the following script demonstrates the use of a tool designed to query
arXiv
, an open-access archive with 2 million scholarly articles across various scientific fields:
metta
! (bind! &arxiv_tool ((py-atom langchain_community.tools.arxiv.tool.ArxivQueryRun))) ! (bind! &arxiv_tool ((py-atom langchain_community.tools.arxiv.tool.ArxivQueryRun))) ! ((py-dot &arxiv_tool invoke) "What's the paper 1605.08386 about?") ;Published: 2011-02-18 Title: Quantum Anticipation Explorer ... ! ((py-dot &arxiv_tool invoke) "What's the paper 1605.08386 about?") ;Published: 2011-02-18 Title: Quantum Anticipation Explorer ...
This example demonstrates how to use the tool individually. The tool can also be used as part of an agent. For this purpose, there is
langchain_openai_tools_agent.msa
in MeTTa-Motto, which utilizes

langchain.agents.AgentExecutor
to execute LLM agents with the use of LangChain tools:
metta
```
! ( import! &self motto ) ! ( import! &self motto ) ! ( import! &self
motto:langchain_agents:langchain_states ) ! ( import! &self
motto:langchain_agents:langchain_states ) ! ( bind! &lst ( py-list ())) ! ( bind!
&lst ( py-list ())) ! (( py-dot &lst append ) (( py-atom
langchain_google_community.GoogleSearchRun ) ! (( py-dot &lst append ) (( py-atom
langchain_google_community.GoogleSearchRun ) ( Kwargs ( api_wrapper (( py-atom
langchain_google_community.GoogleSearchAPIWrapper )))))) ( Kwargs ( api_wrapper ((
py-atom   langchain_google_community.GoogleSearchAPIWrapper )))))) ! (
set-langchain-agent-executor  &lst ) ! ( set-langchain-agent-executor  &lst ) ! (
llm ( Agent motto/langchain_agents/langchain_openai_tools_agent.msa )( user "What is
the name of the airport in Cali, Colombia?" )) ;"The name of the airport in Cali,
Colombia is Alfonso Bonilla Aragón International Airport." ! ( llm ( Agent
motto/langchain_agents/langchain_openai_tools_agent.msa )( user "What is the name of
the airport in Cali, Colombia?" )) ;"The name of the airport in Cali, Colombia is
Alfonso Bonilla Aragón International Airport."
```
The Google search tool is used here to get the answer to the user's question. The script includes the import of the
langchain_states.metta
file, which contains helper functions to create and store prompts, construct
langchain.agents.create_tool_calling_agent
, and set parameters for
langchain.agents.AgentExecutor
.

Advantages of MeTTa-Motto

Using MeTTa-Motto, we can process user messages with LLMs to create new knowledge bases or extend existing ones. These knowledge bases can be further processed using MeTTa expressions and then utilized in MeTTa-Motto to solve various tasks. For example, let's consider an agent defined in the file named
some_agent.msa
:
metta
```
! ( Response ! ( Response ( _eval ( _eval ( llm ( Agent ( chat-gpt )) ( llm ( Agent
( chat-gpt )) ( system "Represent natural language statements as expressions in
Scheme. ( system "Represent natural language statements as expressions in Scheme. We
should get triples from statements, describing some relations between items. We
should get triples from statements, describing some relations between items.
Relation of location  should be  represented with 'location' property. Relation of
location  should be  represented with 'location' property. Relation of  graduated
from (or studies)  should be presented as 'educated_at' property. Relation of
graduated from (or studies)  should be presented as 'educated_at' property. For
example, the sentence 'New York City is  located at the southern tip of New York
State' should be transformed to For example, the sentence 'New York City is  located
at the southern tip of New York State' should be transformed to ( \" New York City
\" location \" New York State \" ). ( \" New York City \" location \" New York State
\" ). 'Lisbon is in Portugal' should be transformed to ( \" Lisbon \" location \"
Portugal \" ). Do not miss quotes. 'Lisbon is in Portugal' should be transformed to
```

(\" Lisbon \" location \" Portugal \"). Do not miss quotes. The sentence 'Ann graduated from the University of Oxford' should be transformed to (Ann educated_at \" Oxford \") The sentence 'Ann graduated from the University of Oxford' should be transformed to (Ann educated_at \" Oxford \") The sentence 'John is studying mathematics at MIT' should be transformed to (John educated_at \" MIT \") The sentence 'John is studying mathematics at MIT' should be transformed to (John educated_at \" MIT \") For questions about place of study we use function study_location, for example: For questions about place of study we use function study_location, for example: The sentence 'Is John studying in the USA?' should be transformed to (study_location John \" USA \") The sentence 'Is John studying in the USA?' should be transformed to (study_location John \" USA \") The sentence 'Did Alan graduate from the University of USA?' should be transformed to (study_location Alan \" USA \") The sentence 'Did Alan graduate from the University of USA?' should be transformed to (study_location Alan \" USA \") The sentence 'Did Mary study in the USA?' should be transformed to (= (study_location Mary \" USA \")) The sentence 'Did Mary study in the USA?' should be transformed to (= (study_location Mary \" USA \")) Return result without quotes." Return result without quotes.")) (messages) (messages)))))))

This agent converts sentences containing location or education-related information into triples, such as
(Ann educated_at "Oxford")
or
("New York City" location "New York State")
. If someone asks about the city or country where the education was received, it converts the question into a MeTTa function. For example: Did Mary study in the USA? will be converted to
(= (study_location Mary "USA")
. Let's define two functions:
is-located
, which checks if
$x
is located in
$y
, and
study_location
, which checks if
$x
studied at a place that is located in
$y
.

metta

```
( = ( is-located $ x $ y ) ( = ( is-located $ x $ y ) ( case ( match &self ( $ x location $ z ) $ z ) ( case ( match &self ( $ x location $ z ) $ z ) ( ( ( %void% False ) ( %void% False ) ( $ z ( if ( == $ z $ y ) True ( is-located $ z $ y ) )) ( $ z ( if ( == $ z $ y ) True ( is-located $ z $ y ) )) ) ) ) ) ) )  ( = ( study_location $ x $ y ) ( = ( study_location $ x $ y ) ( case ( match &self ( $ x educated_at $ z ) $ z ) ( case ( match &self ( $ x educated_at $ z ) $ z ) ( ( ( %void% False ) ( %void% False ) ( $ z ( if ( == $ z $ y ) True ( is-located $ z $ y ) )) ( $ z ( if ( == $ z $ y ) True ( is-located $ z $ y ) )) ) ) ) ) ) )
```

Then, using the

some_agent.msa
, we can add certain relations to the meta space based on the provided facts in natural language, and verify certain facts about the place of study.
metta
! (import! &self motto) ! (import! &self motto) (Fact "Harvard is located in Massachusetts state") (Fact "Harvard is located in Massachusetts state") (Fact "Massachusetts state is located in United States") (Fact "Massachusetts state is located in United States") (Fact "Ann graduated from Harvard.") (Fact "Ann graduated from Harvard.") ! (match &self (Fact $ fact) ! (match &self (Fact $ fact) (let $ expr (llm (Agent some_agent.msa) (user $ fact)) (let $ expr (llm (Agent some_agent.msa) (user $ fact)) (add-atom &self $ expr)) (add-atom &self $ expr)))) ! (get-atoms &self) ! (get-atoms &self) ! (llm (Agent some_agent.msa) (user "Did Ann study in the United States?")) ;True ! (llm (Agent some_agent.msa) (user "Did Ann study in the United States?")) ;True
This is a straightforward example demonstrating the potential of Metta-Motto to integrate MeTTa functionality with the capabilities of LLMs.
Basics of Functional Programming in MeTTa

Coming Soon