MeTTa Specification

Alexey Potapov

August 31, 2021

0 Scope

This document specifies the internal representation and semantics of the MeTTa (Meta Type Talk) programming language. The surface syntax can be different, although we use its default version (close to the internal representation) for convenience.

1 Introduction

MeTTa is designed as a language of the cognitive architecture OpenCog Hyperon focused on grounded knowledge representation and reasoning with inference control and neural-symbolic integration in mind.

2 Notation

Basic atomic elements of the MeTTa language are **symbols**, which are distinguished by their names. Sequences (strings) of arbitrary (Unicode) characters are considered as valid names, although the shallow syntax should not necessarily support this arbitrariness. Some symbols are built-in and thus reserved (using their names will result in referring to the corresponding built-in symbols with their specific interpretation, so they cannot be used as custom symbols).

Symbols can be **grounded**, which means that they are references to some external pieces of data (which can contain executable code as well).

Symbols are composed into **expressions**, which can be represented as trees. The surface syntax closest to this representation is nested lists in the Lisp style, which is used in this document. However, the surface syntax can be different (e.g., Haskell-like). Any expression is a valid term by default unless there are constraints imposed by **types** (described later).

Not nested expressions are lists, and multiple expressions can contain the same symbols, e.g.

```
(possesses Sam balloon)
(has-color balloon blue)
(likes Sam blue)
```

while nested expressions can contain identical subexpressions, e.g.

```
(possesses Sam (blue balloon))
```

```
    (likes Sam (blue balloon))
```

There are different ways to represent such overlapping expressions simultaneously. One way is to deduplicate all identical subexpressions. This leads to a metagraph representation. Another way is to keep trees without any deduplication (yet different occurrences of the same name should be considered the same symbol). However, how these expressions are stored is outside the scope of this document.

Expressions constitute a **program**, if they are placed in the container, which we will refer to as an **Atomspace** that supports a special form of pattern matching (described below). Loading a program's code corresponds to inserting its expressions into the Atomspace. After insertion, these expressions can be considered as subexpressions of the Atomspace, which can be considered as an expression corresponding to the whole program. It is important that we should be able to distinguish expressions, which are root subexpressions of the Atomspace.

The Atomspace as a container or as an expression cannot contain itself or be a part/ subexpression of itself. However, the Atomspace can be represented as a grounded symbol, so it can contain expressions that refer to it and can access its content. We will use **`@root`** for this here.

MeTTa can use different Atomspaces to store its programs and to retrieve expressions via pattern matching, although each Atomspace should support a certain basic API.

3 Basic Types

Symbols are gradually typed. Types impose restrictions on expressions accepted as valid terms. The built-in symbol **`:`** (colon) can be used to build expressions definitionally relating symbols to their types. For example, expression `(: Sam Human)` allows `Sam` to appear in such places in expressions that expect symbols typed as `Human` or if the type restrictions are not indicated (in the vein of gradual typing, this can be represented via assigning type `?` to any symbol without type information).

The built-in symbol **`Type`** is intended to indicate that some symbol can be used as a type, e.g. `(: Human Type)`.

The next question is how to assign types to expressions. Since any expression is allowed by default, it is also natural to form expressions from type symbols. For example, if `(: blue Color)` and `(: balloon Object)`, then `(blue balloon)` is supposed to be a valid expression of the type `(Color Object)`. One can treat this expression and its type as a tuple and a product type correspondingly.

However, we would like some expressions to have types different from "products" (type expressions, in which symbols are just replaced with their types). For example, if we look at `(+ (+ 1 2) 3)`, we will see that it is more natural to describe the type of `+` as something that allows constructing any expression of the form `(+ x y)`, where, say, `(: x Int)` and `(: y Int)`, and the type of the whole expression is also `Int`.

The built-in symbol `->` (arrow) serves for describing types of symbols that are allowed to appear in certain expressions and that determine the resulting type of the expression. Namely, if we define `(: a (-> B A))`, then the expression `(a b)` will be allowed if symbol or expression `b` has either type `B` or undefined, and the type of the whole expression `(a b)` will have either type `A` or undefined correspondingly. Symbol `a` will also be allowed in expressions of the form `(h a)`, if `h` has type `(-> (-> B A) C)` (for any type `C`) or undefined.

There is an ambiguity if `(a b)` is treated as a function application or as a pair (in terms of the expression type; the process of interpretation of such expressions is specified later). We could treat any expression as a function application as in some other languages. If we would like to define a pair instead of a function application, we then needed to write `((pair a) b)` (or, more generally `((quote a) b)`). `pair` could be a polymorphic function of type `(-> A (B -> ((Pair A) B)))`, or for better readability `A -> B -> Pair A B`, for any `A` and `B` (specification of parametric and dependent types will be provided below). Then, `Pair` could also have an arrow type like `Pair : Type -> Type -> Type`. However, even functional languages, which interpret any expression as functional application of the first element of the expression to the rest elements, don't go that far to interpret expressions like `(: a A)` also as functional applications (although one could imagine an arrow type for `:` in a certain metalanguage). In MeTTa we consider this an unnecessary wrapping, since its main purpose is in declarative knowledge representation rather than functional programming. Thus, MeTTa doesn't interpret, say, `(5 5)` as an application, and doesn't prohibit such expressions. Restrictions on expressions are imposed, only when their first element has an arrow type. The programmer should be aware of this conditional imposition of typing rules, because it may catch fewer mistakes. However, although `(5 5)` is a well-typed expression, `(+ (5 5))` is not, because `+` in our example expects `Int`, not `(Int Int)`, thus mistakes in expressions will be caught when these expressions are used in other expressions with arrow-typed symbols.

Another approach to avoid this ambiguity would be to treat all expressions as a product type, so even `(a b)` with `(: a (-> B A))` and `(: b B)` would have type `((-> B A) B)`. Then, it would be necessary to explicitly indicate each function application. This application would be a part of metalanguage, because while its signature would be clear, i.e. `(-> ((-> B A) B) A)`, it wouldn't be a normal expression, because it should have a special rule for its type derivation. Thus, in any approach there will be exceptional expressions with specific rules for their type derivation. In MeTTa, there are just two common rules for deriving expression types, the choice between which is determined by the type of the first symbol of the expression.

MeTTa doesn't have built-in symbols to describe types of expressions symmetric to `(a b)`. For example, we could have `(: a (<- A B))` that allows expressions like `(b a)` for `(: b B)` with the type `A` for the whole expression. Similarly, one cannot describe an arrow-like type for functional symbols in the middle of expressions. Although such expressions as `(Sam likes balloon)` are not prohibited in principle, the type system discourages from using them by not providing corresponding function type constructors, which restrict well-formed expressions. Derivation of the type of `(Sam likes balloon)` can still be done via definitional equalities over types (described below), but only in run-time.

However, MeTTa supports the arrow type with an arbitrary number of arguments. For example, one can define

```
(: has-color (-> Object (-> Color Bool)))
```

so that the expression `((has-color balloon) blue)` would be valid. Indeed, in this case, `(has-color balloon)` is of `(-> Color Bool)` type, and `((has-color balloon) blue)` is of `Bool`. At the same time, one can define

```
(: has-color (-> Object Color Bool))
```

so that `(has-color balloon blue)` is valid and has type `Bool`. Currying and uncurrying are not done automatically and should be performed explicitly. It should be noted that both these types are different from another possible definition `(: has-color (-> (Object Color) Bool))`, for which the right expression will be `(has-color (balloon blue))`.

The arrow type with only the return type is also allowed, e.g. `(: f (-> Int))`. Such functions are typically considered in functional programming as receiving an element of the unit type, for example, `f : () -> Int`. This makes mathematical sense and allows writing `f ()` as a legal expression. In MeTTa, it naturally occurs that if we consider expressions just structurally without their interpretation, then if we define `(: f (-> Int))`, then expression `(f)` will be of type `Int` without introducing additional type derivation rules.

It should be noted that MeTTa doesn't have a separate notion and syntax for type constructors. When a typed symbol is introduced, it can be considered as a constructor of this type. For example,

```
(: Human Type)
(: Socrates Human)
(: Plato Human)
```

can be considered as roughly equivalent to data `Human = Socrates | Plato` . This allows MeTTa to support insertion of new knowledge base entries in a disentangled and modular way in runtime.

The following

```
    (: HList Type)
    (: Nil HList)
    (: Cons (-> Human HList HList))
```

is roughly equivalent to

```
    data HList = Nil | Cons Human HList
```

Indeed, `(Cons Plato (Cons Socrates Nil))` will type-check and will have `HList` type. That is, arrow-typed constructors (including recursive type definitions) fit to this representation.

The following

```
    (: Mortal (-> Human Type))
```

can be considered as a simple dependent type. Indeed, expressions like `(Mortal Socrates)` will type-check and will have `Type` type. One can treat such types as propositions, which are true if they are inhabited. Thus, judgmentally introducing a witness `(: mortalSocrates (Mortal Socrates))` can be interpreted as prior knowledge (not a proposition to be proven) that `Socrates` is `Mortal`. Whether to follow the propositions-as-types view in this traditional representation or not is outside the scope of this specification, but let us note that these type definitions are roughly equivalent to the following (in Idris):

```
    data Mortal : Human -> Type where
        mortalSocrates : Mortal Socrates
```

`(: mortalSocrates (Mortal Socrates))` can indeed be considered as a type constructor. Equivalents for `List a` or `allHumansAreMortal : (h : Human) -> Mortal h` will be provided later, because they need a special class of symbols, variables.

Grounded types are supported by MeTTa. On a syntactic level, grounded types are described via regular expressions, so the parser can recognize them and pass matched sequences of symbols to the corresponding grounded type constructor. Registering custom grounded types is a part of the Atomspace API, which can be accessible both from the MeTTa program itself (since the reference to the Atomspace is available as a predefined grounded symbol) or from other languages, for which this API is available. A concrete specification of this API is out of the scope of this documentation. The following code is provided only as an example. The Python code

```
    parser.register_token("\\d+(\.\\d+)",
                          "Float",
                          lambda token: ValueAtom(float(token)))
```

could tell the parser than any symbol name that matches `"\\d+(\.\\d+)"` should be inserted into the Atomspace as a grounded symbol that wraps the data (Python object) returned by the call `float(token)`, and the type of the resulting grounded symbol is `Float`. Then, an expression like `(1.0 5)` while being loaded to the Atomspace will be automatically converted to an expression composed with grounded symbols, which wrap binary representations of `1.0` and `5`, and have type `Float`. More complex objects can be wrapped by grounded symbols similarly. Regular expressions can correspond not to patterns but to individual symbols that can be introduced independently as belonging to the same type (i.e. as its constructors). These objects can also wrap code execution, and thus can have an arrow type, e.g. `(-> Float Float Float)` for `+`, that will make expressions like `(+ 1.0 5)` valid, and which types will be derived using common mechanisms. In general, already introduced grounded symbols and grounded types do not differ (on the level of allowed expressions) from pure symbols and types.

TBD: non-deterministic types.

4 Static Pattern Matching

A container for expressions is an Atomspace if it is equipped with a pattern matcher, which extracts expressions for this container that can be matched with the given query expression. Although pattern matching is a part of Atomspace, it is the base of MeTTa, which sets the requirements to it, so it is a part of this specification.

The pattern matcher utilizes a special class of symbols, namely, **variables**. Variables can be free or bound to certain expressions. The binding of variable symbols can change over time, so it is a part of language semantics. Variables can appear both in expressions, stored in the Atomspace, and in queries. Here, we will distinguish variables by prefix `$`, e.g. `$x`. A (sub)expression in the Atomspace is considered to be matched with a query, if the expression and the query can be made identical by non-contradictory replacement of variables with certain subexpressions.

For example, the query `($p Sam $o)` can be matched against `(possesses Sam balloon)` and `(likes Sam (blue stuff))` with variables `$p` and `$o` bound to `possesses` and `balloon`, and `likes` and `(blue stuff)` correspondingly, but it cannot be matched against `(has-color balloon blue)` or `(likes Marry Sam)`. Similarly, the query expression `($p Sam $o)` could be the expression stored in the Atomspace, while `(likes Sam (blue stuff))` could be the query, and they still would be matched with the same variable bindings with the only difference that these variables would belong to an expression in the knowledge base instead of the query. We will denote the result of successful query as a matched expression from the Atomspace with indicated variable bindings. For example,

```
    (possesses Sam balloon) | $p <- possesses, $o <- balloon
```

will indicate that the retrieved expression is `(possesses Sam balloon)` for the query `($p Sam $o)`, and the bound query variables `$p` and `$o`. At the same time,

```
    ($p Sam $o) | possesses -> $p, balloon -> $o
```

will indicate that `($p Sam $o)` is the retrieved expression, and `$p` and `$o` are the bound expression variables.

The same variable can appear several times in an expression. In this case, it should be bound to the same subexpression. Otherwise, the matching will not be successful. For example, if `($x equals $x)` is an expression stored in the Atomspace, then the queries `(Sam equals Sam)` and `((blue stuff) equals (blue stuff))` will be successfully matched against it with `$x` bound to `Sam` and `(blue stuff)` correspondingly, while `(Sam equals Mary)` will not be matched against `($x equals $x)`.

Variables in queries and expressions are treated asymmetrically. Binding of query variables is prioritized. For example, matching query `($y equals $z)` against expression `($x equals $x)` will be successful with `$y` and `$z` both bound to $x, which will remain unbound (free), i.e. the query result will be `($x equals $x) | $y <- $x, $z <- $x`. On the contrary, matching query `($x equals $x)` against expression `($y equals $z)` will result first in binding `$x` to `$y`, and then in binding `$z` to `$x` (since `$x` is already bound), i.e. `($y equals $z) | $x <- $y, $x -> $z`.

However, binding a variable to a non-variable symbol or expression is prioritized. Matching `(Sam equals $z)` against `($x equals $x)` will result in binding `$x` to `Sam`, and `$z` also to `Sam` (via `$x`) independently on which of them is a query.

Variables can appear on different levels of expressions. The following pairs cannot be matched:

```
    ($x (a $y)) and (b a)
    ($x (a $x)) and (b ($y a))
    ($x $x) and ($y ($y))
```

while these pairs can

```
    ($x $y) and (b (a b))
    ($x (b $x)) and (a ($y a))
    ($x $z) and ($y ($y))
    ($x $x) and ($y ($z))
```

In the last case, binding a variable to an expression (even with variables) is prioritized, i.e. `$x` will be bound to `($z)` as well as `$y` that will also be bound to `($z)` via `$x`.

Special symbols are used to form complex query expressions. One such symbol is `,` (comma). It is used to form multiple queries with shared variables, which should be satisfied simultaneously. For example, the following query

```
    (, (possesses Sam $object)
        (likes Sam ($color stuff))
        (has-color $object $color))
```
is treated not as one expression that should be found in the knowledge base, but as three query expressions, with shared variables, which should be bound simultaneously. For example, they can be matched against `(possesses Sam balloon)`, `(likes Sam (blue stuff))`, `(has-color balloon blue)`, if these expressions are found in the program.

TBD a query language that supports more complex queries up to their chaining ("folding / unfolding on metagraphs")

The pattern matcher supports types. Although type expressions are also MeTTa expressions, type information is taken into account separately for convenience. For example, when the query `(: (Cons Socrates Nil) HList)` is executed, it can be successful even if this full expression wasn't added to the program (Atomspace), but if just `(Cons Socrates Nil)` was added. It should be underlined that if this expression wasn't added, the query will not be successful independent from the possibility to construct this expression in accordance with defined types. Although any constructable expression is considered as existent in type theories, the pattern matcher doesn't try to check constructability. Constructing an object of a given type as a proof of the proposition corresponding to this type is not the task of the pattern matcher. Its task is to retrieve stored expressions matchable with queries. For example, the query `(: (Cons $x Nil) HList)` would retrieve nothing if no expression of the form `(Cons _ Nil)` was actually added to the Atomspace despite type constructors `(: Nil HList)` and `(: Cons (-> Human HList HList))` could help guessing that `$x` could be bound to instances of `Human`.

However, the pattern matcher uses type definitions to perform type checking. For example, `(: (Cons Socrates Nil) Human)` will retrieve nothing, even if `(Cons Socrates Nil)` was added without type annotations, but types of `Nil` and `Cons` were described (although it should be noted that type inference is undecidable in dependent type theories, so type information is propagated through equalities only in runtime – see the details below). Whether type annotations are derived by the pattern matcher each time it is called, or type annotations are derived for all subexpressions at the moment of inserting the expression into the Atomspace, and whether these type annotations are stored as ordinary expressions or separately are the questions of the Atomspace and the pattern matcher implementations, and are outside the scope of this document.

Types of variables can be specified in queries. For example, imagine `cons` and `nil` without type definitions. Types of such symbols are ?, and types of expressions of the form `(cons _ _)` will also be ?. Both `(cons Socrates nil)` and `(cons mortalSocrates nil)` will type-check. However, we can form the query `(, (: $h Human) (cons $h $t))` that will match against the first expression, but not against the second expression.

Variables can be used to retrieve types of symbols or expressions. For example, `(: (Cons Socrates Nil) $t)` will bind `$t` to `HList` (given corresponding type definitions). However, let us underline once again that if there is no `(Cons Socrates Nil)` in the Atomspace, the query result will be empty. Such a query should be understood not as an attempt to infer the type of an arbitrary expression, but as an attempt to find the given expression or pattern in the Atomspace.

Variables can also appear in type definitions (like in other MeTTa expressions). Consider the following

```
      (: List (-> Type Type))
      (: Nil (List $a))
      (: Cons (-> $a (List $a) (List $a)))
```

`(: (Cons Socrates Nil) (List Human))` will type-check.

The pattern matcher is directly accessible from programs in MeTTa via using `match` grounded symbol. The process of evaluation of grounded symbols will be described below. Here, we just indicate that evaluation of match will return a special structure wrapped into a grounded symbol of `Match` type, which incorporates retrieved expressions with information about variable symbol bindings.

`transform` is another interface function of the pattern matcher, which accepts one additional (in comparison to `match`) argument, which defines the output expression. That is, instead of retrieving matched expressions with variable bindings, `transform` returns (the list of) new expressions (typically, constructed on the base of bound variables). For example, `(transform (possesses Sam $o) $o)` will return `(' balloon ball)`, if the query is matched against `(possesses Sam balloon)` and `(possesses Sam ball)`. Since `transform` returns a MeTTa expression instead of a grounded object, it is more convenient for composing it with other expressions.

Enriched expressions. We treat types as enrichment of expressions, because, as mentioned above, typing `(: a A)` is not treated as an ordinary expression from the pattern matcher standpoint. If `(: a A)` was added to the Atomspace as a separate expression or subexpression, it will still be matched against just `a` appearing in the same place, e.g. the query `(e a)` will be successfully matched against `(e (: a A))` the knowledge base entry. If `a` was added to the Atomspace without type annotations, the query `(: a $t)` will still be successful (with `$t` bound to `?`). Types of symbols can also be provided separately. For example, if the program was

```
      (: a A)
      (e a)
```

the query `(e (: a $t))` will be successful and will bind `$t` to `A`. That is, type annotations are treated by the pattern matcher as supplementing expressions with additional information rather than modifying their structure, that we refer to as expression enrichment.

Other enrichments are possible. For example, each expression can be enriched with vector embedding or attention value. In the current version of specification, such enrichments are considered as language extensions, which are done via using an Atomspace and its pattern matcher implementation that extends the basic Atomspace API. Custom enrichment of expressions from MeTTa programs themselves is not defined in the current version, although it is supposed that in the future versions it will be done via grounded symbols that are executed on insertion to the Atomspace rather than in runtime.

Remark. The legacy OpenCog adopts a somewhat different approach. It separates `Atoms` and `Values` and attaches `Values` to `Atoms`. However, this separation comes more from the difference in storage and retrieval (e.g. `Values` are not indexed). At the same time, the legacy OpenCog has various built-in `Atom` types, which are secretly grounded atoms. For example, `NumberNode` and `AddLink` work similar to grounded symbols in MeTTa. Also, `TruthValues` are a special kind of `Values`. They could be considered as an expression enrichment in MeTTa, but it may not be sensible to ascribe truth value to every expression. For example, what would be the meaning of `(+ 2 2)` to be true or false (to a certain degree)? It would be better to talk about truth values not of arbitrary expressions, but of propositions. We will describe this in the next section. Thus, we consider enrichments as pieces of information that should accompany every expression, but which should not accompany themselves if they were also represented as ordinary expressions (one could try to describe this by a Kleisli category or, equivalently, a monad as well as an enriched category, but we don't pursue a strict category-theoretic interpretation here). For example, assigning types to typing expressions (e.g. `(: (: a A) ...)`) ad infinium doesn't seem useful.

5 Equalities

Built-in symbol `=` is used by the MeTTa interpreter to chain pattern matcher queries (specified below). It can be an element of expressions with the restriction corresponding to the type signature `(: = (-> $t $t Type))`. It is a typical way of introducing propositional equality (as a type) in dependently typed languages with the only constructor `Refl x = x`. In MeTTa, however, we treat all definitional equalities as such constructors. Consider a simple example:

```
    (= (double $x) (+ $x $x))
```

Query `(= (double 2) $y)` will bound `$y` to `(+ 2 2)`. This is an ordinary query, although it can be seen that if we want to evaluate `(double 2)`, we need the interpreter to construct and chain such equality queries. This process will be specified in the next sections.

Now, consider query `(= (double 2) 4)` that should also be a valid query. After binding `$x` to `2`, the pattern matcher will have to match `(+ 2 2)` and `4`. A direct static matching of these two expressions should be unsuccessful. However, we want the interpreter to be able to unify them up to propositional equality. Since the pattern matcher doesn't solve equalities, it cannot match these expressions directly, but can determine that `(= (double 2) 4)` is matchable against `(= (double $x) (+ $x $x))` if `(+ 2 2)` and `4` are not necessarily directly matchable, but equal (i.e. can be transformed to the same expression by a sequence of equality matches). The pattern matcher will not recursively call itself. Instead, it will return a conditional match, in which the necessity for further unification of `(+ 2 2)` and `4` is indicated in the form of expression `(unify (+ 2 2) 4)` to be further evaluated. Let us underline that the query `(+ 2 2)` by itself will not be matched (even conditionally) against `4` or another expression, which can be propositionally equal to it. The pattern matcher is intended not to evaluate expressions, but only to match them.

Such a query as `(= (double $y) 4)` will be matched against `(= (double $x) (+ $x $x))` conditioned on `(unify (+ $x $x) 4)`. Whether the latter can be successful or not depends on the definition and equalities of `+`.

Consider the following expression
```
(: dbleq (= (double $x) (+ $x $x)))
```
It is valid since equality is a type. At the same time, we don't need to provide an element of the equality type in order to be able to pattern-match against the equality expression itself as we have seen above. However, one may not just pattern-match against this equality, but form propositions about it. If `(= (double $x) (+ $x $x))` has `Type` type, what can it be equal to? In type theory, all not empty types as propositions are propositionally equal to each other in the sense that they are all true, i.e. they are all equal to a unit type. In MeTTa, we don't insist on such interpretations, and MeTTa doesn't convert type inhabitance into type equality automatically. Instead, one can introduce a custom symbol `(: True Type)` and write
```
(= (= (double $x) (+ $x $x)) True)
```
Then, the query
```
(= (= (double  2) (+  2  2))  $y )
```
will be successful and bind `$y` to `True`. It is better to think about these constructions as purely syntactic for now. Similar to the above, one can form the query
```
(= (= (double 2) 4) $y)
```
which will be matched against `(= (= (double $x) (+ $x $x)) True)` conditioned on `(unify (+ 2 2) 4)`.

In MeTTa, we can write more complex equalities for type symbols and expressions. Consider the example

```
      (: TCons Type)
      (: Cons TCons)
      (: (List $a) Type)
      (= (TCons $a (List $a)) (List $a))
```
In the same way, we can write
```
      (: :: T::)
      (= ($a T:: (List $a)) (List $a))
```
However, with the use of equalities on types, types of expressions like `(Cons Socrates Nil)` or `(Socrates :: Nil)` will not be automatically derived. Static type checking doesn't process equalities (including equalities on types). The types of these expressions will remain `(TCons Human (List $a))` and `(Human T:: (List $a))`. However, the query `(= (TCons Human (List $a)) $t)` will bind `$a` to `Human` and `$t` to `(List Human)`. A more complex query
```
      (, (: (Socrates :: Nil) $texp)
         (= $texp $t))
```
should also work.

It should be noted that `Type` can be populated with any symbols and expressions, for which arbitrary equalities can be introduced. Let us consider one specific example,
```
      (: TV (-> Float Type))
      (: Entity Type)
      (: Socrates Entity)
      (: Sam Entity)
      (: Human (-> Entity Type))
      (= (Human Socrates) (TV 0.9))
      (= (Human Sam) (TV 0.7))
      (: And (-> Type Type Type))
      (= (And (TV $p1) (TV $p2)) (TV (* $p1 $p2)))
```
These are all valid expressions. Then, we can execute the query
```
      (= (And (Human Socrates) (Human Sam)) $tv)
```
which will be matched against the last expression in the code above conditioned on unifications `(Human Socrates)` with `(TV $p1)` and `(Human Sam)` with `(TV $p2)`, which can be further done by equality queries. The process of interpretation of expressions like `(And (Human Socrates) (Human Sam))`, which yields `(TV 0.63)` for this expression, is described below.

6 One step of evaluation

Grounded symbols as stand-alone expressions (for example, `5.0`, `“String”`, or `@root`) are always evaluated to themselves.

Expressions starting with symbols of non-functional (not arrow) types are interpreted as tuples and are evaluated to themselves, e.g. evaluation of `(5 +)` will return the expression itself as the final result. Evaluation of subexpressions of tuples is not invoked by the interpreter automatically (TBD in the lazy computations settings, which can be controlled by directives). The type `?` is considered functional, and thus expressions starting with untyped symbols will be treated as function applications.

Evaluation of expressions, whose first element is a grounded symbol of a functional type, starts with evaluation of its subexpressions unless the grounded symbol expects arguments of `Expression` metalanguage type. The result of evaluation of subexpressions is passed (on the next step of evaluation) to `execute` method of the grounded functional symbol, if this result has the expected type and its metalinguistic type is not `Expression`. For example, if the functional grounded symbol expects an argument of a grounded type, the subexpression should be evaluated to the grounded symbol of the corresponding grounded type instead of remaining an expression, even if its derived type corresponds to the expected type (e.g. `(+ 5 5)` can have derived type `Int`, but its metalinguistic type is `Expression`, so it will be evaluated first before passing to a superexpression, say, `(+ (+ 5 5) 2)`). Otherwise, the expression with application of the grounded symbol evaluates to itself.

Evaluation of other expressions including stand-alone not grounded symbols consists in forming and executing an equality query. For evaluating expression `e`, the query `(= e $t)` is formed and executed. If the query returns an empty result, the expression is evaluated to itself. Otherwise, each query result will be evaluated on the next steps, and their results will be returned as the results of evaluation of `e`. The pattern matcher retrieves only root expressions. It doesn't retrieve any matchable subexpression appearing in any context. Retrieving any subexpression can be supported by the pattern matcher, but is not used by the MeTTa interpreter. TBD scopes.

Precise match of `(= e $t)` will produce an expression bound to `$t` as a new evaluation target, which will be evaluated in an ordinary way on the next step. Conditional matches, in turn, will require taking into account one more expression to be equated with the query result. If this result is an expression of application of a grounded functional symbol, this grounded symbol will be executed directly first. Otherwise, an equality query based on both expressions will be constructed. Above, we encountered two examples of conditional matches. Evaluation of `(unify (Human Socrates) (TV $p1))` will proceed by constructing the equality query `(= (Human Socrates) (TV $p1))`, which execution will yield a precise match.

Evaluation of `(unify (+ 2 2) 4)` will depend on how `+` is defined. If it is a grounded symbol, it will be executed first, yielding `4`. Equating two grounded symbols (of the same type) requires the grounded type to support equality check (which is supposed by the grounded atoms API by default). Thus, equating `(= 4 4)` on the next step will successfully eliminate the condition of the initial conditional match. TBD support for solving equalities by grounded symbols of functional type.

Equating expressions in a conditional match can result in another conditional match that will become the next evaluation target. For example, evaluating

```
(unify (+ (S Z) (S Z)) (S (S Z)))
```

given equalities

```
(= (+ $x Z) $x)
(= (+ $x (S $y)) (+ (S $x) $y))
```

will be done via query `(= (+ (S Z) (S Z)) (S (S Z)))` that will be matched against the second equality expression with binding `$x` to `(S Z)`, `$y` to `Z` conditioned on `(unify (+ (S (S Z)) Z) (S (S Z)))` (which, in turn, will be unconditionally successful on the next step).

7 Interpretation

The MeTTa interpreter works with a number of expressions being evaluated, which are stored in a separate container (local Atomspace). An atomic interpretation step consists in forming a query to the program (knowledge base) Atomspace, which can be a pattern matching query or request to evaluate an application of a functional grounded symbol. The output of this one step of evaluation can be either the final result or an intermediate result, which is placed into the local Atomspace for further evaluation.

The local Atomspace can be considered as a queue by default, although the policy of selecting the next evaluation target can be different including concurrent evaluation of multiple targets as well as control of the choice of the next target by the program itself (TBD).

Consider one complete example. Let the knowledge base Atomspace (program) contain the following expressions

```
(: Nat Type)
(: Z Nat)
(: S (-> Nat Nat))
(: + (-> Nat Nat Nat))
(= (+ $x Z) $x)
(= (+ $x (S $y)) (+ (S $x) $y))
```

Consider `(+ (S Z) (S Z))` as an interpretation target. The interpreter will perform the following steps

1. Query `(= (+ (S Z) (S Z)) $r)` is executed resulting in binding `$r` to `(+ (S (S Z)) Z)`, which will be the next interpretation target.

2. Query `(= (+ (S (S Z)) Z) $r)` is executed resulting in binding `$r` to `(S (S Z))`.
3. Query `(= (S (S Z)) $r)` fails, so `(S (S Z))` is evaluated to itself, and no next interpretation target is constructed. `(+ (S Z) (S Z))` is evaluated to `(S (S Z))`.

Evaluation of expression `(= (+ Z Z) Z)` will also start with forming the query `(= (= (+ Z Z) Z) $r)`, which will fail, since there is no expression of this form in the Atomspace. Thus, one should distinguish evaluating MeTTa expressions and executing pattern matching on them. In particular, `(match (= (+ Z Z) Z))` is another valid expression, and its evaluation will be successful, because match is a grounded symbol of type `(-> Expression Match)`, and it will be executed without constructing an equality query.

One can also evaluate `(match (= (+ Z $v) (S Z)))`. Its execution on the first step of interpretation will result in a conditional match against `(= (+ $x (S $y)) (+ (S $x) $y))` with `Z->$x, $v<-(S $y)` conditioned on `(unify (S Z) (+ (S Z) $y))`. Conditional matching will form the next evaluation target.

At the same time, if we write

```
(= (= (+ $x Z) $x) True)
(= (= (+ $x (S $y)) (+ (S $x) $y)) True)
```

without having internal equalities as separate expressions, then evaluation of expressions like `(+ Z Z)` will not produce desirable results, because equalities like `(= (+ $x Z) $x)` will not be top-level expressions. One should have both

```
(= (+ $x Z) $x)
(= (= (+ $x Z) $x) True)
```

to achieve the desirable behavior.

One can introduce (together with the initial definitions in our example) the equality

```
(= (= $a $b) (transform (= $a $b) True))
```

This will cause evaluation of `(= (+ Z Z) Z)` to be as follows

1. Query `(= (= (+ Z Z) Z) $r)` is executed resulting in binding `$r` to `(transform (= (+ Z Z) Z) True)`.
2. Evaluation of `(transform (= (+ Z Z) Z) True)` proceeds as direct execution of `transform` as a grounded symbol, which successfully matches `(= (+ Z Z) Z)` against the knowledge base entry `(= (+ $x Z) $x)` and thus returns `True`.

Alternatively, one can introduce `(= (: $t Type) (transform (: $w $t) True))` meaning that an element of `Type` is evaluated to `True` if it is not empty (there is an element of this type).

Let us consider the full interpretation of `(And (Human Socrates) (Human Sam))` for the program

```
(: TV (-> Float Type))
(: Entity Type)
(: Socrates Entity)
(: Sam Entity)
(: Human (-> Entity Type))
(= (Human Socrates) (TV 0.9))
(= (Human Sam) (TV 0.7))
(: And (-> Type Type Type))
(= (And (TV $p1) (TV $p2)) (TV (* $p1 $p2)))
```

The following steps will be performed:

1. Query `(= (And (Human Socrates) (Human Sam)) $r)` is executed resulting in binding `$r` to `(TV (* $p1 $p2))` conditioned on `(unify (Human Socrates) (TV $p1))` and `(unify (Human Sam) (TV $p2))`.
2. Query `(= (Human Socrates) (TV $p1))` is successfully executed binding `$p1` to `0.9`.
3. Query `(= (Human Sam) (TV $p2))` is successfully executed binding `$p2` to `0.7`.
4. Conditions are satisfied and the next query is formed `(= (TV (* 0.9 0.7)) $r)`.
5. TBD if there are no special rules forcing evaluation of TV subexpressions or eager evaluation directive is not used, `(TV (* 0.9 0.7))` will be the final result.

Let us consider one more example:

```
(= (if True $then $else) $then)
(= (if False $then $else) $else)
```

The interpretation process is organized in such a way that evaluating an if-expression will not require simultaneously evaluating expressions passed to `if` as both `$then` and `$else`. One of these expressions will be evaluated on the next steps of interpretation, when the first argument passed to `if` will be unified either with `True` or `False`.

Totality of functions is not checked by default. Functions and type constructors are not explicitly distinguished. The practical difference between them is absence or presence of definitional equalities, depending on which expressions of their application will be evaluated to themselves or will be processed through equality queries. Thus, non-total functions can act as functions and as type constructors simultaneously. This approach is adopted due to multiple reasons. The main reason is that new symbols can be introduced in runtime and gradually. Introduction of a new type constructor would break the totality of the previously defined functions. Also, knowledge can be (and typically is) incomplete, and a symbol that appears as a type constructor initially can gain some definitional equalities later. TBD a directive is available to require function totality, which forces any symbol of functional type to either have no definitional equalities (i.e. to be a type constructor) or to be defined on the whole domain.

Let us consider one example of a non-total function:

```
      (= (do-if True $then) $then)
```

Evaluating this expression with the first argument equal to `True` will result in evaluating `$then`. Otherwise, the expression will remain unchanged. We can manipulate the expressions like `(do-if (has-color Sun Green) (jump))`, so they can be considered sensible as expressions, but not reducible (unless we define additional equalities for transforming them).

Of course, keeping unreduced expressions of not total functions applications doesn't correspond to what is customary in pure functional programming, and can be undesirable, when we do want to work with total functions. However, the idea behind MeTTa is to be a flexible meta-language supporting different possibilities, so totality is considered as an additional possible constraint rather than obligatory feature.

7 Non-determinism

`match` and `transform` can return multiple results, each or which will form the basis of subsequent chained queries. Say, `(transform (likes Sam $x) $x)` is a perfectly valid query from the knowledge base standpoint, which can return multiple results. If we don't just directly execute queries, but evaluate them as MeTTa expressions, their results will be evaluated further making the interpreter essentially non-deterministic. In general, queries of the form `(transform (= (f $x) value) $x)` that invert functions will be non-deterministic.

By default, MeTTa also doesn't insist on uniqueness of functions, so we don't need to resort to `transform` to invoke non-determinism. Equalities

```
      (= (bin) 0)
      (= (bin) 1)
```

can look contradictory, because normally they would imply that `0=1`. However, in MeTTa equality is not symmetric. One should read `(= (bin) 0)` as that `(bin)` can be transformed to `0` in the process of evaluation, but `0` cannot be evaluated to `1`. Evaluation of `(bin)` will be done via constructing query `(= (bin) $r)` yielding both `0` and `1` as the result of non-deterministic evaluation, while `0` and `1` will be evaluated to themselves.

Also, if one wants to indicate that some statement is true and reasons about it, this should be done by equalities in Type. Of course, if one writes

```
      (= (= (bin) 0) True)
```

```
(= (= (bin) 1) True)
(= (And True True) True)
(= (= $a $b) (And (= $c $a) (= $c $b)))
```
then evaluation of `(= 0 1)` will yield `True`. The semantic mistake here is in the first two lines of the program.

As we considered above, we can use compound truth value (or even anything else) instead. It will make more sense if we write:
```
(= (= (bin) 0) (TV 0.5))
(= (= (bin) 1) (TV 0.5))
```
We should interpret these as causal probabilities of getting `0` or `1` from evaluating `(bin)`. Such interpretation is not enforced, and one can introduce fuzzy logic rules, which can make perfect sense for some programs, knowledge bases, and applications (but which can lead to a conclusion that `0=1` to a certain degree in this case that can also be not senseless in certain contexts), although it can be better to introduce custom symbols for this.

Thus, by default, equalities suppose directed paths in the space of expressions (and functions have an arrow type, not equivalence). This can be used to define causal generative models. For example, one can define
```
(= (gen Nil) Nil)
(= (gen (Cons $h $t))
   (Cons (bin) (gen $t)))
```
that will generate all binary lists of the same length as the input list. Then,
```
(= (smult Nil Nil) 0)
(= (smult (Cons $x $t1) (Cons $b $t2))
   (+ (* $x $b) (smult $t1 $t2)))
```
will calculate the sum of products of elements of two lists. Then, one can define the solver for the subset sum problem as
```
(= (subsum $xs)
   (match (= (smult $xs (gen $xs)) 0))
)
```
TBD let-bindings

Evaluation of, say, `(subsum (Cons 4 (Cons -1 (Cons -3 Nil))))` will start with the corresponding equational query leading to evaluation of `(match (= (smult $xs (gen $xs)) 0))` with `$xs` bound to `(Cons 4 (Cons -1 (Cons -3 Nil)))`. Execution of this query will be successful conditioned on unification of `(gen $xs)` with `(Cons $b $t2)` and `(+ (* $x $b) (smult $t1 $t2))` with `0`. The first unification will non-deterministically give two bindings for `$b`, while the second unification will further unroll recursive `smult`. Finally, when the whole binary list is generated and the subset sum is calculated, the result of unification will be successful, if this sum will be equal to `0`. Such matches will be kept, while others will be rejected.

Although equational non-determinism can be used to solve some tasks directly, it is not proposed as a universal inference method, because it can easily lead to bloating of the local Atomspace let alone inefficient enumeration of all possibilities. However, it can be a useful primitive for implementing more advanced inference algorithms possibly involving inference control. Consider the simplest use:

```
(: sampler (-> Expression Nat ?))
(= (sampler $e (S $n))
   (sampler $e $n))
(= (sampler $e $n)
   (sample $e))
```

which will produce the specified number of `(sample $e)` as concurrent interpretation targets (supposing that `sample` doesn't interpret `$e` non-deterministically, but samples one of its possible evaluations).

As with the absence of a restriction on the totality of functions by default, the restriction on the uniqueness of functions is considered additional and optional.