AI-DSL Technical Report 2024

Nil Geisweiller

June 19, 2025

Abstract

Contents

Chapter 1

Introduction

During the middle of 2023 it became apparent that MeTTa [2] could soon be used instead of Idris [5] for program synthesis. Around the end of 2023 a general purpose chainer [6] was developed and improved throughout 2024 and we thus began experimenting with it to synthesize AI service compositions. In this document we will go over the work that was done in order to accomplished such a feat. It can be summarized as follows:

- Develop and improve a general purpose chainer in MeTTa supporting backward, forward and in fact *omniward* chaining and can handle dependent types.
- Develop a SingularityNET Market Place crawler to gather information about AI services and convert that information into MeTTa specifications.
- Extensively experiment with AI service composition using the aforementioned chainer. Various AI service composition representations (including lambda abstraction and combinatory logic) were explored, as well as the tractability of the corresponding synthesis processes.
- Prototype an AI-DSL ontology in MeTTa.
- Implement a MeTTa to DOT converter to graphically display synthesized AI service compositions.

At the end of this process we finally managed to efficiently synthesize the English to Chinese Song test case described in [8] (an add-on to the technical report of 2022 [9]). By itself this is of course very promising. It should be mentioned though that, in order to make the synthesis tractable, only the services involved in that composition were considered in the knowledge base fed to the chainer. Attempting to perform such synthesis while considering all available AI services will be the subject of the next round of work on the AI-DSL.

Chapter 2

Type Driven Program Synthesis in MeTTa

In this chapter we will explain how program synthesis can be done in MeTTa and go over the various backward chainers we have prototyped during 2023 and 2024. This will come handy for Chapter 4 which goes over a number of experiments on AI service compositions based on these chainers. Why backward chaining, you may ask? Because it allows to go from theorems to axioms. In the context of AI service composition, a theorem would be the formal specification of an overall AI service composition, such as

> *Turn any English song into a corresponding Chinese one*

and the axioms would be the formal specification of every AI service involved in the composition, such as

- *Convert speech to text.*
- *Translate English to Chinese.*
- *Turn Audio into MIDI.*
- ...

Then the backward chainer would take the overall specification and combine existing AI services to fulfill it altogether. Going forward could be useful as well but for a different type of queries, such as

> *Given all these AI services, what can you come up with by combining them?*

and the forward chainer would combine them to form random, albeit valid, compositions and provide their formal specifications. Then there is everything between forward and backward chaining, what I like to call *omni* chaining.

For instance, the query could provide an incomplete specification of the overall composition, alongside some AI services which should be involved, and let the omni chainer come up with completions of such specifications using the provided AI services. My prototypes cover all of these possibilities, but in this report we will only focus on backward chaining because that is what I exclusively used for the AI-DSL work so far.

2.1 Curried Backward Chainer

Let us begin with the Curried Backward Chainer, the simplest of them all. The curried backward chainer is simple because it assumes that there is only one way to construct terms, using unary function application. In this case a term can be either

- a constant,
- or a term applied to a term.

A constant could represent a value such as `42`, or a function like a standalone AI service such as `speech2text`. In an application, the first term must correspond to a unary function, such as `foo`, and the second term must correspond to its argument, such as `42`, resulting in an application such as `(foo 42)`. Emulating n-ary application can be done by considering higher order unary functions. For instance `(+ 1 2)` can be represented in curried format by `((+ 1) 2)`. Contrary to most functional programming languages, currying is not handled automatically in MeTTa, thus a non-curried version of `+` would be typed

```
(-> Number Number Number)
```

while a curried version would be typed

```
(-> Number (-> Number Number))
```

In the curried backward chainer, all functions are assumed to be unary and that is how currying is handled. As per the Curry-Howard correspondence, functions on the programming side correspond to inference rules on the logical side, so our curried backward chainer can also handle synthesizing proofs, not just programs. The AI-DSL actually uses both sides at the same time producing AI service compositions containing programs and proofs. Why on earth would we want that is explained in detail in Chapter 4.

The MeTTa implementation of the curried backward chainer can be found in [7] is given in full below

```
;; Curried Backward Chainer type signature
(: bc (-> $a                            ; Knowledge base space
          Nat                           ; Maximum depth
          $b                            ; Query
          $b))                          ; Result

;; Base case
(= (bc $kb $_ (: $prf $thrm))
   (match $kb (: $prf $thrm) (: $prf $thrm)))

;; Recursive step
;; Unary proof application
(= (bc $kb (S $k) (: ($prfabs $prfarg) $thrm))
   (let* (;; Recursive call on function
          ((: $prfabs (-> $prms $thrm))
           (bc $kb $k (: $prfabs (-> $prms $thrm))))
          ;; Recursive call on argument
          ((: $prfarg $prms)
           (bc $kb $k (: $prfarg $prms))))
     ;; Query with holes filled
     (: ($prfabs $prfarg) $thrm)))
```

Let us walk over that code.

1. The type signature takes

   - A knowledge base containing the axioms, which can in fact be viewed as the rewrite rules as well. In a way the knowledge base contains the description of the logic the backward chainer is going to operate on.
   - A maximum depth corresponding to the maximum depth of the syntax tree that the backward chainer is allowed to produce.
   - A query of the form

     ```
     (: TERM TYPE)
     ```

     indicating that `TERM` is of type `TYPE`. The query may contains free variables representing holes that the backward chainer must fill. For instance if the query is

     ```
     (: $prg (-> Number String))
     ```

     there is one big hole, `$prg`, in place of the term, indicating that the backward chainer must find a program with the type signature

     ```
     (-> Number String)
     ```

     The same backward chainer can be used to infer a type if the hole is placed on type. For instance the query

     ```
     (: (foo 42) $type)
     ```

indicates that the backward chainer must infer the type of `(foo 42)`. Holes can be placed anywhere and at any depth across the term and the type of the query, such as

```
(: (+ $val) (-> $input Number))
```

and the backward chainer must attempt to fill the holes regardless. Each result returned will be the query itself with the holes filled. If more than one result exists then the backward chainer will return a superposition of results.

2. The base case

```
(match $kb (: $prf $thrm) (: $prf $thrm)))
```

is simply a match query over the knowledge base. If the query is an axiom, then it returns it. If it matches several axioms, then it returns a superposition of all matches.

3. The recursive step occurs if the query is of the form

```
(: ($prfabs $prfarg) $thrm)
```

corresponding to a unary application, and the depth is greater than zero, enforced by having to match the depth argument `(S $k)`. If the backward chainer enters that call, then it breaks up the query into two subqueries:

   (a) one to discover `$prfabs`, the function,

   (b) the other to discover `$prfarg`, the argument.

`$prfabs` stands for *proof abstraction*, reflecting the idea that it is a function that takes a proof in input and outputs a proof, merely corresponding to a regular function on the programming side of the Curry-Howard isomorphism. And `$prfarg` stands for *proof argument*, reflecting the idea that it is an argument provided to a proof abstraction. On the logical side of the Curry-Howard correspondence, you can roughly think of `$prfabs` as being an inference rule, while `$prfarg` being the proof of a premise of that inference rule. I say roughly because `$prfabs` may not just be an inference rule, it can be more general than that, a proof function that takes in input a proof and outputs a proof, or *proof abstraction* as I like to call it, which you can think as a composite inference rule. The broken up query to discover the function is

```
(bc $kb $k (: $prfabs (-> $prms $thrm)))
```

ordering the backward chainer to look for a proof abstraction that, if given a proof of a premise, `$prms`, to be discovered as well, then it outputs a proof of `$thrm`, the conclusion. The broken up query to discover the argument is

```
(bc $kb $k (: $prfarg $prms))
```

ordering the backward chainer to look for a proof of the premise `$prms`.

One may notice that, unlike function definitions in regular functional programming languages, the base case is not constrained by its depth. In the base case of this MeTTa program, `$_` does not mean *otherwise*, it means *any time*. The non-determinism of MeTTa allows both the base case and the recursive step to be taken simultaneously. The resulting effect is that a call of `bc` can bottom down at any depth up to the maximum depth, producing proof trees of any size up to the maximum depth. Let us now provide an example. First, let us fill the knowledge with a theory

```
;; Knowledge base
!(bind! &kb (new-space))
!(add-atom &kb (: 42 Number))
!(add-atom &kb (: foo (-> Number String)))
!(add-atom &kb (: bar (-> String Bool)))
!(add-atom &kb (: . (-> (-> $b $c) (-> (-> $a $b) (-> $a $c)))))
```

That theory expresses that `42` is a number, provides two casting functions, one from `Number` to `String`, called `foo`, and the other one from `String` to `Bool`, called `bar`. Finally, it provides a higher order composition operator `.` also called the *Bluebird* combinator in [**?**]. Given that theory we can now call the backward chainer with a few queries. For starter, let us infer the type of 42.

```
;; Infer the type of 42
!(bc &kb Z (: 42 $type))
```

which outputs

```
[(: 42 Number)]
```

Next, let us synthesize terms of type `String`

```
;; Synthesize terms of type String
!(bc &kb (S Z) (: $prg String))
```

which outputs

```
[(: (foo 42) String)]
```

The depth for this query must be at least 1, represented by `(S Z)` as `Nat`, because the term to be synthesized is a function application requiring to enter the recursive step of `bc` at least once. Finally, let us synthesize all unary functions that outputs a Boolean value.

```
;; Synthesize all functions that output a Boolean value
!(bc &kb (S (S Z)) (: $prg (-> $intput Bool)))
```

which outputs the superposition of two solutions

```
[(: ((. bar) foo) (-> Number Bool)),
 (: bar (-> String Bool))]
```

one turning a number into a Boolean value, `((. bar) foo)`, the other one turning a string into a Boolean value, `bar`.

To help you understand what is going I have printed the trace of the `bc` call of the last query. `bc-bas` corresponds to the base case entry, `bc-rec` corresponds to the recursive step entry. The knowledge base argument is missing from the trace to be more concise. I have manually reconstructed the tree representing the recursive calls and added some comments. Note that the tree does not show the distinction between non-determinism across branches and regular functional evaluation along one branch. I hope the trace conveys what is going on in spite of that omission. Obviously it would be nice if MeTTa could offer a tool to automatically display such trace and show such distinctions.

```
| ;; Original call, base case, succeeds (match bar)
|-(bc-bas (S (S Z)) (: $prg (-> $intput Bool)))
| ;; Original call, recursive step
|-(bc-rec (S (S Z)) (: ($prfabs#219 $prfarg#220) (-> $intput Bool)))
  | ;; First recursive call on function, base case, fails
  |-(bc-bas (S Z) (: $prfabs#219 (-> $prms#222 (-> $intput Bool))))
  | ;; First recursive call on function, recursive step
  |-(bc-rec (S Z) (: ($prfabs#730 $prfarg#731) (-> $prms#222 (-> $intput Bool))))
  | | ;; Second recursive call on function, base case, succeeds (match .)
  | |-(bc-bas Z (: $prfabs#730 (-> $prms#733 (-> $prms#222 (-> $intput Bool)))))
  | | ;; Second recursive call on argument, base case, succeeds (match bar)
  | |-(bc-bas Z (: $prfarg#731 (-> $b#1364 Bool)))
  | ;; First recursive call on argument, base case, succeeds (match foo)
  |-(bc-bas (S Z) (: $prfarg#220 (-> $intput String)))
  | ;; First recursive call on argument, recursive step
  |-(bc-rec (S Z) (: ($prfabs#2202 $prfarg#2203) (-> $intput String)))
    | ;; Second recursive call on function, base case, fails
    |-(bc-bas Z (: $prfabs#2202 (-> $prms#2205 (-> $intput String))))
```

Upon the second recursive call on function, entering the base case

```
(bc-bas Z (: $prfabs#730 (-> $prms#733 (-> $prms#222 (-> $intput Bool)))))
```

the successful match of the query against

```
(: . (-> (-> $b $c) (-> (-> $a $b) (-> $a $c))))
```

creates bindings which are passed upstream to the caller (the first recursive call on function). As a result, by the time the second recursive call on argument enters the base case

```
(bc-bas Z (: $prfarg#731 (-> $b#1364 Bool)))
```

the premise `$prms#733` has been substituted by `(-> $b#1364 Bool)`. This comes from `(-> $b $c)` because `$c` was unified with `Bool` while attempting to match `(-> $intput Bool)` against `(-> $a $c)`.

If at this point what is going on is still unclear, I recommend to run the code, query by query, while tracing the function calls. To that end I have included a file [**?**]

`curry-backward-chainer-example.metta`

containing the code described above, wrapped in `trace!` calls. As its name indicates `trace!` is a MeTTa primitive to trace MeTTa code.

2.2 Uncurried Backward Chainer

It is not always convenient to manipulate curried expression, in that case, extending the curried backward chainer to support more than unary functions can be done by adding more entries in the backward chainer definition. Specifically, right below the unary proof application recursive step

```
;; Unary proof application
(= (bc $kb (S $k) (: ($prfabs $prfarg) $thrm))
   ...)
```

one may simply add

```
;; Binary proof application
(= (bc $kb (S $k) (: ($prfabs $prfarg1 $prfarg2) $thrm))
   (let* (;; Recursive call on function
          ((: $prfabs (-> $prms1 $prms2 $thrm))
           (bc $kb $k (: $prfabs (-> $prms1 $prms2 $thrm))))
          ;; Recursive call on first argument
          ((: $prfarg $prms1)
           (bc $kb $k (: $prfarg1 $prms1)))
          ;; Recursive call on second argument
          ((: $prfarg $prms2)
           (bc $kb $k (: $prfarg2 $prms2))))
     ;; Query with holes filled
     (: ($prfabs $prfarg1 $prfarg2) $thrm)))
```

to support uncurried binary functions. Or

```
;; Ternary proof application
(= (bc $kb (S $k) (: ($prfabs $prfarg1 $prfarg2 $prfarg3) $thrm))
   (let* (;; Recursive call on function
          ((: $prfabs (-> $prms1 $prms2 $prms3 $thrm))
           (bc $kb $k (: $prfabs (-> $prms1 $prms2 $prms3 $thrm))))
          ;; Recursive call on first argument
          ((: $prfarg $prms1)
           (bc $kb $k (: $prfarg1 $prms1)))
          ;; Recursive call on second argument
          ((: $prfarg $prms2)
           (bc $kb $k (: $prfarg2 $prms2)))
          ;; Recursive call on third argument
          ((: $prfarg $prms3)
           (bc $kb $k (: $prfarg3 $prms3))))
     ;; Query with holes filled
     (: ($prfabs $prfarg1 $prfarg2 $prfarg3) $thrm)))
```

to support uncurried ternary functions, etc. One may write a MeTTa macro (which is just a regular MeTTa program) to generate such code for any given arity. Although since the arities of the functions we manipulate for reasoning are usually low, we have not found the need to do that so far.

2.3 Embed Inference Rules

Another possible extension of the backward chainer is to embed its axioms and inference rules directly in its code. For instance the theory given in example in Section 2.1 we be directly implemented as the following specialized backward chainer

```
;; Base cases
(= (bc $_ (: 42 Number)) (: 42 Number))
(= (bc $_ (: foo (-> Number String))) (: foo (-> Number String)))
(= (bc $_ (: bar (-> String Bool))) (: bar (-> String Bool)))

;; Recursive step
;; Function composition
(= (bc (S $k) (: (. $prfarg1 $prfarg2) (-> $a $c)))
   (let* (;; Recursive call on first argument
          ((: $prfarg1 (-> $b $c))
           (bc $kb $k (: $prfarg1 (-> $b $c))))
          ;; Recursive call on second argument
          ((: $prfarg2 (-> $a $b))
           (bc $kb $k (: $prfarg2 (-> $a $b)))))
      (: (. $prfarg1 $prfarg2) (-> $a $c))))
```

In this example the entire theory is embedded in the backward chainer implementation, but one can also write a hybrid backward chainer with some rules being generic, and some being embedded in the code. One advantage of embedding the theory directly in the backward chainer implementation is that some axioms or rules can be given some special treatments. Examples of such implementations will be shown in Chapter **??**.

2.4 Dependent Types

Simply explained, *dependent types* [**?**] allow to use values inside types. An example of what can be done with dependent types that is often given is a vector data structure where the size of the vector is specified within the type itself. Such definition may in Idris look like

```
-- Vector type parameterized by element type and size
Vect : a -> Nat -> Type

-- Build a vector by repeating a given element n times
repeat : a -> (n : Nat) -> (Vect a n)
```

Given that definition one may build the following

```
(repeat "abc" 42) : (Vect String 42)
```

corresponding to a vector of 42 strings of `"abc"`. One can then pursue to write operators manipulating vectors allowing consistency checking on their sizes to take place at compile time. For instance `append` would have the following type signature

```
-- Append one vector of size n with another one of size m
append : (Vect a n) -> (Vect a m) -> (Vect a (n + m))
```

Note that the vector size inside the output type is `(n + m)`.

As of today, dependent types are not supported by the built-in type checker of MeTTa. Luckily for us however, only a few modifications need to be operated to have the backward chainer support dependent types in MeTTa.

2.4.1 Rule Format for Dependent Types

First, the format of an inference rule must be changed from

```
(-> PREMISE CONCLUSION)
```

to

```
(-> (: ARGUMENT PREMISE) CONCLUSION)
```

where `ARGUMENT` would typically appear inside `CONCLUSION`, the *dependent* part of dependent types. An example of such inference rule would be

```
(: translate
   (-> (: $src-lang NaturalLanguage) (: $dst-lang NaturalLanguage)
       (-> (: $_ (TextIn $src-lang)) (TextIn $dst-lang))))
```

This inference rule represents an actually AI service from the SingularityNET market place that translates a text in some source language to an equivalent text in some destination language. The first two arguments of the type signature of the service are the source and destination languages

```
(: $src-lang NaturalLanguage)
(: $dst-lang NaturalLanguage)
```

Because the argument terms, `$src-lang` and `$dst-lang`, are specified in the type signature, they can be passed as parameters to the other following types

```
(TextIn $src-lang)
(TextIn $dst-lang)
```

where `TextIn` is a parameterized type representing a text in a given language. One may notice the use of `$_` representing a variable that is not subsequently used to create dependencies[1]. That is because in the current format, unlike

[1]Please be aware that in MeTTa `$_` is a regular variable and does behave like an underscore in a functional programming language like Haskell. Thus multiple occurrences of `$_` within the same scope still will point to the same variable.

in Idris, specifying the argument associated to the input of an arrow type is mandatory. This may eventually become optional to have more concise type signatures. One may also notice that the rule is a mixture of curried and uncurried arguments. The first two arguments are uncurried while the last one is curried. This is perfectly fine and comes from a convention that has been adopted in some experiments, which is that arguments of AI services corresponding to hyper-parameters are uncurried, while those corresponding to data being processed are curried. The reason for this convention is explained in detail in Chapter **??**.

2.4.2 Backward Chainer for Dependent Types

With all that said, modifying the backward chainer to support dependent types is as trivial as it may be. The code is identical to the backward chainers presented above but the queries must follow the format presented in Subsection 2.4.1. The full implementation for the modified curried backward chainer is given below.

```
;; Dependently Typed Curried Backward Chainer type signature
(: bc (-> $a                              ; Knowledge base space
          Nat                             ; Maximum depth
          $b                              ; Query
          $b))                            ; Result

;; Base case
(= (bc $kb $_ (: $prf $thrm))
   (match $kb (: $prf $thrm) (: $prf $thrm)))

;; Recursive step
;; Unary proof application
(= (bc $kb (S $k) (: ($prfabs $prfarg) $thrm))
   (let* (;; Recursive call on function
          ((: $prfabs (-> (: $prfarg $prms) $thrm))
           (bc $kb $k (: $prfabs (-> (: $prfarg $prms) $thrm))))
          ;; Recursive call on argument
          ((: $prfarg $prms)
           (bc $kb $k (: $prfarg $prms))))
     ;; Query with holes filled
     (: ($prfabs $prfarg) $thrm)))
```

Only two lines have changed, the query of the recursive call on function has gone from

```
(: $prfabs (-> $prms $thrm))
```

to

```
(: $prfabs (-> (: $prfarg $prms) $thrm))
```

that is all. An example of using such dependently typed backward chainer to prove properties about programs can be found in **??**. The chainer used for most of the AI-DSL experiments is based on it, with the additional support for uncurried and embedded rules. We are almost done regarding chaining, but there is one last extension we need to cover because some experiments were conducted with it, the support for lambda abstraction.

2.5 Lambda Abstraction

In λ-calculus, lambda abstraction, or λ abstraction, is a way to construct functions. On the proof side of the Curry-Howard correspondence, λ abstraction is a way to construct proof functions, or what I like to call *proof abstractions*, a function that takes a proof in argument and outputs a proof. If the argument is the proof of a certain hypothesis H, and the output is a proof of a conclusion C, then the proof abstraction is a function that computes a way to go from a proof of H to a proof of C, which can be denoted with the arrow type as $H \to C$. Function construction is the dual of function application. Just like function construction can be viewed as hypothesis introduction, function application can be viewed as modus ponens. That is, given a body $c : C$, containing a free variable x of type H, one can use lambda abstraction to construct the following implication

$$\lambda x.c : H \to C$$

Likewise, applying such proof abstraction to proof $h : H$, results in proof

$$((\lambda x.c)\ h) : C$$

which, one may note, perfectly emulates the behavior of the modus ponens rule. Thus one may consider that the backward chainer presented so far is essentially a generic implementation of modus ponens. It makes sense therefore that one would want to support its dual operation, lambda abstraction. There is an alternative though, which is to use *Combinatory Logic*. A combinator is a higher order function that takes functions in inputs and output yet more functions. Some combinator set, such as S, K and I are known to be sufficiently expressive so that one can built any function out of them. That is how they can represent an alternative to λ abstraction. Our most successful experiments were in fact conducted with combinators (albeit another set than S, K and I), while most of our experiments conducted with λ abstraction failed due to the excessive and uncontrollable combinatorial explosion resulting from it. The backward chainer implementations presented so far already support combinaroty logic, one just needs to define the set of combinators to be used in the theory handed to the backward chainer. For λ abstraction it is a bit more complicated and the backward chainer needs to be modified specifically for that purpose. Let us show below how.

2.5.1 Backward Chaining with Lambda Abstraction

The main idea needed to have the backward chainer support λ abstraction is that, when going backward, unpacking a λ abstraction must introduce knowledge in the theory about the type of the variable previously scoped by the λ. Formally, this can be represented as the following inference rule

$$\frac{\Gamma, x : s \vdash f : t}{\Gamma \vdash \lambda x.f : s \to t} \ (\lambda \text{ Introduction})$$

meaning that if $\lambda x.f$ is typed $s \to t$ in theory Γ, then, as we go backward, to be able to infer the type of f, we need to add information about the type of x into Γ. Indeed, as the backward chainer keeps unpacking f, it will eventually meet x, then query the theory about its type. If the type is missing the backward chainer will fail to reach all the axioms, and thus fail to infer the type of f. In other words, for λ abstraction to be properly supported, $x : s$ must temporarily become an axiom of the theory as backward chaining is taking place.

Going back to MeTTa, in our backward chaining implementations the theory is stored in a space for efficiently matching inference rules and axioms. So far the theory was static, only defined once before launching the backward chainer and left unchanged while running. To support λ abstraction however, we need to be able to dynamically and locally modify such space in every non-deterministic branches as the backward chainer unpacks λ abstractions. Unfortunately, MeTTa does not support locally modifying spaces, though an issue [**?**] has been created and it will hopefully be supported in the foreseeable future. In the meantime we have implemented our own data structure alongside a custom match operator mimicking what a locally modifiable space would do. It is not efficient, our custom match operator is linear in complexity, but it gives us the tools that we need to experiment with λ abstraction, as well as any other reasoning schemes involving dynamically modifying a theory, such as modal and contextual reasoning, which are not presented in this document but are also of capital importance.

2.5.2 Locally Modifying Space

Below is the code of our MeTTa implementation of a locally modifiable space.

```
;; Define match', like match but takes a list of terms as space
(: match' (-> (List Atom) $a $a $a))
;; Base case, empty space
(= (match' Nil $pattern $rewrite) (empty))
;; Base case, match with the head
(= (match' (Cons $h $t) $p $r) (let $p $h $r))
;; Recursive step, match with the tail
(= (match' (Cons $h $t) $p $r) (match' $t $p $r))
```

So a space is merely a list of MeTTa terms. Once more, non-determinism is put to use to iterate over the list while returning the superposition of matches. For the sake of completeness the code of `List` is provided below

```
;; Define List data type and constructors
(: List (-> $a Type))
(: Nil (List $a))
(: Cons (-> $a (List $a) (List $a)))
```

2.5.3 Encoding Variables as De Bruijn Indices

In the backward chainer, MeTTa variables take the role of holes in the query. In order to enable the backward chainer to manipulate variables in the proofs and programs being synthesized while avoiding any possible confusion with holes, proof and program variables are encoded as De Bruijn indices.

```
;; Define DeBruijn type and constructors
(: DeBruijn Type)
(: z DeBruijn)                          ; Zero
(: s (-> DeBruijn DeBruijn))            ; Successor
```

Thus `z` represents the first De Bruijn index, `(s z)` represents the second, `(s (s z))` represents the third, and so on. One may notice that `DeBruijn` is isomorphic to `Nat`

```
;; Define Nat type and constructors
(: Nat Type)
(: Z Nat)
(: S (-> Nat Nat))
```

It still needs to be provided though so that the backward chainer can make the distinction between a natural number and a variable.

2.5.4 Lambda Abstraction in MeTTa

With all that in hand we can finally provide the full implement of the backward chainer supporting λ abstraction. Let us start with its type signature.

```
;; Define Backward Chainer with Lambda Abstraction
(: bc (-> $a                            ; Knowledge base space
          (List $b)                     ; Environment
          DeBruijn                      ; De Bruijn Index
          Nat                           ; Maximum depth
          $b                            ; Query
          $b))                          ; Result
```

Compared to the backward chainer implementations above, two extra arguments are introduced:

- An environment, typed `(List $b)`, holding the dynamic part of the theory, solely dedicated to store typing relationships of variables dynamically introduced by the backward chainer as it encounters λ abstractions. While the knowledge base space still statically holds the rest of the theory.

- A De Bruijn index, typed `DeBruijn`, to be used as variable for the next encountered λ abstraction.

For instance, let us say the backward chainer is called on

```
(bc $kb Nil z (S (S Z)) (: (\ z (+ z z)) (-> Number Number)))
```

corresponding to a type checking query of `(\ z (+ z z))`, an anonymous function which doubles a number, against the type `(-> Number Number)`. Initially, the environment is empty, `Nil`, meaning no variable has been introduced so far, or equivalently, no lambda abstraction was encountered so far. The De Bruijn index to be introduced next is `z`, matching the index in the λ abstraction of the query. Subsequently, a recursive call is crafted by the backward chainer to

1. unpack the λ abstraction,
2. insert typing information about `z` in the environment, here `(: z Number)`,
3. increment the De Bruijn index, thus `(s z)`, for a future potential lambda abstraction.

The resulting call looks like

```
(bc $kb (Cons (: z Number) Nil) (s z) (S Z) (: (+ z z)) Number))
```

At this point the backward chainer still needs to unpack `(+ z z)` and go one level deeper in the recursion to reach the axioms

```
(bc $kb (Cons (: z Number) Nil) (s z) Z (: z Number))
```

which it can, since the query `(: z Number)` unifies with one the axioms in the environment, which happens to be a singleton in this example. The full implementation of the backward chainer supporting λ abstraction is given below.

```
;; Base cases
;; Match the knowledge base
(= (bc $kb $env $idx $_ (: $prf $thrm))
   (match $kb (: $prf $thrm) (: $prf $thrm)))
;; Match the environment
(= (bc $kb $env $idx $_ (: $prf $thrm))
   (match' $env (: $prf $thrm) (: $prf $thrm)))

;; Recursive steps
;; Proof application
(= (bc $kb $env $idx (S $k) (: ($prfabs (: $prfarg $prms)) $thrm))
   (let* (((: $prfabs (-> $prms $thrm))
           (bc $kb $env $idx $k (: $prfabs (-> $prms $thrm))))
          ((: $prfarg $prms)
           (bc $kb $env $idx $k (: $prfarg $prms))))
     (: ($prfabs (: $prfarg $prms)) $thrm)))
;; Proof abstraction
(= (bc $kb $env $idx (S $k) (: (\ $idx $prfbdy) (-> $prms $thrm)))
   (let (: $prfbdy $thrm)
     (bc $kb (Cons (: $idx $prms) $env) (s $idx) $k (: $prfbdy $thrm))
     (: (\ $idx $prfbdy) (-> $prms $thrm))))
```

I am hoping that by now the mechanism is clear. If it is not, feel free to play with the implementation and the examples provided in [**?**, **?**].

We have covered everything we need to know in order to synthesize AI services compositions in a type driven manner using MeTTa. An obvious prerequisite though, is to have the knowledge of AI services and their type specifications, in the first place. The next Chapter will look into the problem of retrieving such information from the SingularityNET Market Place.

Chapter 3

Representing the SingularityNET Market Place in MeTTa

3.1 SingularityNET and the Blockchain

Explaining in detail how the SingularityNET Market Place works is beyond the scope of this document, but let me nonetheless attempt to give you a high level view of how the SingularityNET Market Place operates on the blockchain. We will focus on the Ethereum blockchain because it is the one that I understand, but I believe what I am about to say applies to the Cardano blockchain side of SingularityNET as well.

A smart contract on the Ethereum blockchain centralizes the process of registering organizations that want to publish services on the SingularityNET Market Place. You may ask, "centralizes", I thought it was decentralized? Yes, it is decentralized in the sense that the contract is ultimately duplicated all over the Ethereum network, but it is centralized in the sense that any modification of it will disperse over the network until a consensus is reached, so that in the end, there is a universal agreement on what this contract contains. Once an organization is registered, it can register AI services via interacting with the same smart contract. Due to the high cost of holding data on the Ethereum blockchain, the smart contract only holds the minimal amount of data, such as organization identifiers, wallets, the list of their services, etc. More data hungry information such as the descriptions of the services are stored in the InterPlanetary File System (IPFS for short), a decentralized, data immutable, content addressable storage system. The smart contract then only needs to point to the right IPFS addresses to retrieve the desired information. Given the immutability of data of IPFS, once a smart contract points to a given address, it is guarantied that the content at that address will not change. To change it, the

smart contract must be modified to point to another address. Thus, all possible modification are funneled through the smart contract. That is not to say that the network cannot change if the smart contract is not changed, for instance a service provider may decide to silently shut down its servers, but the description itself of the SingularityNET Market Place should be immune to these external circumstances. That is of course assuming that both the Ethereum blockchain and the IPFS networks are collectively given the care to operate as intended. In particular, for IPFS, there needs to be at least one node in the network containing the data referenced by the SingularityNET smart contract.

3.2 Crawling the SingularityNET Market Place

The goal of this process is to gather information about all AI services on the SingularityNET Market Place and convert that information into a format that the backward chainer can use to synthesize AI service compositions.

To that end, a bash script driving the SingularityNET CLI [4] has been implemented to crawl the SingularityNET Market Place, see [**?**]. First, it gathers all organizations and their services in JSON format. Second, it converts that into MeTTa. It can afford to be implemented in bash because the heavy lifting of accessing the information inside the smart contract is outsourced to the SingularityNET CLI, and then reading that information in JSON format is outsourced to jq [1], a command-line JSON processor. At the end of the process we are left with a folder tree structure of JSON files organized by organizations and services, and a monolithic MeTTa file containing the definitions of all organizations, their services and their associated type signatures.

Such MeTTa file can then be imported in a space and used to reason about organizations and their services, and in particular for what is our main interest here, to reason about their compositions. One can find examples of such monolithic dumps in MeTTa format in [**?**]. Let us give a few snippets of such a dump for illustrative purposes. For instance, the data type to represent an organization is defined in MeTTa as follows:

```
;; Define Organization type
(: Organization Type)

;; Define Organization constructor
(: MkOrganization
   (->
       String ; org_name
       String ; org_id
       String ; org_type
       Description ; description
       (List Assets) ; assets
       (List Contact) ; contacts
       (List Group) ; groups
       Organization))
```

One may immediately notice that the format is consistent with the format of axioms and inference rules the backward chainer expects, a collection of typing relationships. An example of the definition of an object of the type `Organization`, here corresponding to the SingularityNET organization, is given below.

```
(MkOrganization
    ; org_name
    "snet"
    ; org_id
    "snet"
    ; org_type
    "organization"
    ; description
    (MkDescription
        ; url
        "https://singularitynet.io"
        ; url content
        null
        ; description
        "We gathered leading minds in machine learning and
         blockchain to democratize access to AI technology.
         Now anyone can take advantage of a global network
         of AI algorithms, services, and agents. The world's
         first decentralized AI network has arrived"
        ; short_description
        "SingularityNET lets anyone create, share, and
         monetize AI services at scale.")
    ; assets
    Nil
    ; contacts
    Nil
    ; groups
    Nil)
```

where `MkDescription` is the data type constructor of `Description`

```
;; Define Description constructor
(: MkDescription
   (->
       String ; url
       String ; url_content
       String ; description
       String ; short_description
       Description))
```

Let us move on to services. The data type to represent a service is defined in MeTTa as follows:

```
;; Define Service type
(: Service Type)

;; Define Service constructor
(: MkService
   (->
       Number ; version
       String ; display_name
       String ; encoding
       String ; service_type
       String ; model_ipfs_hash
       String ; mpe_address
       (List Group) ; groups
       ServiceDescription ; service_description
       (List Contributor) ; contributors
       (List Medium) ; media
       (List String) ; tags
       Service))
```

An example of the definition of an object of the type `Service` is given below, here corresponding to the Machine Translation service from Native Intelligence

```
(MkService
    ; version
    1
    ; display_name
    "Machine Translation"
    ; encoding
    "proto"
    ; service_type
    "grpc"
    ; model_ipfs_hash
    "QmcQBTx9qZcTVijZFZSdiesdwtwFoywvkEbtYSkF9bmxwi"
    ; mpe_address
    "0x5e592F9b1d303183d963635f895f0f0C48284f4e"
    ; groups
    Nil
    ; service_description
    (MkServiceDescription
        ; url
        "https://github.com/iktina/neural-machine-translation"
        ; url content
        null
        ; description
        "<div>The service receives text in one language and
         returns a translation of the submitted text in another
         language. Translation is possible for 204 languages.\n
```

```
      You can pass text or the URL of a text file. The input
      text or text file in the URL must contain up to
      4500-5000 characters.</div>"
     ; short_description
     "The service receives text in one language and returns
      a translation of the submitted text in another
      language. Translation is possible for 204 languages. ")
; contributors
Nil
; media
Nil
; tags
(Cons "text2text"
      (Cons "text"
            (Cons "multilanguage"
                  (Cons "translation"
                        (Cons "nmt"
                              (Cons "nlp" Nil)))))))
```

Already this information could be reasoned upon by the backward chainer due to being formatted as typing relationships. However, none of the examples given so far expresses anything about input and output types of services. This information is encoded as Protocol Buffers (or Protobuf for short) specifications. The SingularityNET CLI allows us to retrieve the Protobuf files associated to each service. Then the Protobuf files can be converted into MeTTa via protobuf-metta [3] a tool specifically created for that purpose. The bash script crawling the market place calls protobuf-metta and populates the resulting MeTTa file with these specifications. Below is an example of such specification pertaining to the same Machine Translation service. First it contains the definitions of the types involved in the service. Right below is its input type

```
;; Define naint.machine-translation.Input type
(: naint.machine-translation.Input Type)

;; Define naint.machine-translation.Input constructor
(: naint.machine-translation.MkInput
   (-> String ; source_lang
       (-> String ; target_lang
           (-> String ; sentences_url
               naint.machine-translation.Input))))
```

The constructor `naint.machine-translation.MkInput` indicates that the Machine Translation service takes 3 arguments in input

1. a string encoding the source language,
2. a string encoding the target language,

3. a URL pointing to the sentence to be translated.

Let us now look at the type of its output

```
;; Define naint.machine-translation.Output type
(: naint.machine-translation.Output Type)

;; Define naint.machine-translation.Output constructor
(: naint.machine-translation.MkOutput
   (-> String ; translation
       naint.machine-translation.Output))
```

As `naint.machine-translation.MkOutput` indicates, the output is a single string containing the result of the translation. In addition to these, the MeTTa file also contains access functions. Together they allow to construct and deconstruct data exchanged between services. So if a service outputs a pair of string and number, and another service wishes to take the string in input without the number, the access function can be used to select only the string. The access function for `naint.machine-translation.Output` is given below.

```
;; Define naint.machine-translation.Output.translation
(: naint.machine-translation.Output.translation
   (-> naint.machine-translation.Output String))
(= (naint.machine-translation.Output.translation
    (naint.machine-translation.MkOutput $translation))
    $translation)
```

Thus it allows to select the translated string given the output of the Machine Translation service. Even though in this case the string is the only output, the access function is still necessary to cast the output into a string. Given these type definitions we can finally declare the service and its corresponding type signature, given below.

```
;; Define naint.machine-translation.translate service method
(: naint.machine-translation.translate
   (-> naint.machine-translation.Input
       naint.machine-translation.Output))
```

This information, data types of inputs and outputs, constructors and access functions, and of course type signatures of the services, is all that the backward chainer needs to compose services together. There is no need to know the implementation details of the services. At least not at this point, although it is conceivable that in the future, knowledge about the implementation of services may become useful, in particular to estimate their performances and costs.

There is one obvious drawback though. The types described by the Protobuf specifications are usually devoid from semantic information, which may be precious to discriminate whether a particular composition is sensible or absurd. A good example is the third argument of the Machine Translation service.

According to Protobuf alone, it is of type string, however one can see in the comment that it is not a string containing the content to be translated but a string pointing to a URL containing the content to be translated. That means that if an afferent service sends a string of text to the Machine Translate service, the latter will fail unless the string of text has been uploaded to a URL and the URL is sent instead. The types alone, at least as provided by the existing Protobuf specifications, are not descriptive enough to catch that sort of errors. In the next section will show how this can be addressed.

3.3 Enriching Type Signatures using the AI-DSL Ontology

The collection of concepts handled in AI applications is especially rich and tied to the real world. For instance to characterize a face recognition algorithm, one needs at most to define what is a face, or at least, in case the definition is left to the AI to discover, to define that there is such a thing as a face. That is what the AI-DSL ontology is meant to provide. Not only symbols corresponding to concepts but also how these concepts relate to each other. How detailed such ontology should be has not been established yet, neither how it should be built. It is clear that it should be sufficiently detailed to be able to discriminate between sensible and absurd AI service compositions most of the time, and it is also clear that its building should be at least partially automated. Additionally, it should likely be decentralized and able to handle multiple versions.

These aspects will have to be addressed eventually, but in this iteration we merely provide a miniature handwritten ontology tailored specifically to discriminate sensible vs absurd compositions in our examples. The ontology itself is formalized using a subtyping relationship described below. The axioms are taken from the Subtyping Wikipedia article [**?**] and translated into MeTTa. Subtyping is represented by

```
(<: S T)
```

indicating that `S` is a subtype of `T`. The full list of axioms and inference rules of the subtyping relationship is provided below in MeTTa.

1. Reflexivity

   ```
   (: STRefl (<: $T $T))
   ```

2. Transitivity

   ```
   (: STTrans (-> (<: $S $T)
                  (<: $T $U)
                  (<: $S $U)))
   ```

3. Contravariant over inputs and covariant over outputs

```
(: STConv (-> (<: $T1 $S1)
              (<: $S2 $T2)
              (<: (-> $S1 $S2) (-> $T1 $T2))))
```

This rule is not as obvious as reflexivity and transitivity, but is also fairly standard. To understand it, it is best to apply it to small examples involving function composition. We leave that to the discretion of the reader. If it still feels counter-intuitive after that it may help to remember that the inferred subtyping relationship is between functions, not between their inputs and outputs.

4. Coercion

```
(: coerce (-> (<: $S $T)
              (-> $S $T)))
```

This last rule is crucial, it expresses that one can automatically coerce an inhabitant of a type into an inhabitant of one of its super types. So for instance, if a function `nbrLimbs` takes an inhabitant of the type `Animal` in input, the `coerce` rule can be used to apply this function to inhabitants of any subtypes of `Animal`, such as `Cat`, as long as it has been proven that `Cat` is indeed a subtype of `Animal`. Such proof obligation corresponds to the first premise of `coerce`. A more formal description of such an example goes as follows. Calling `nbrLimbs` on the cat `Choupette` can be done with

```
(nbrLimbs (coerce CA Choupette))
```

where `CA` is a proof that `Cat` is a subtype of `Animal`, formally

```
(: CA (<: Cat Animal))
```

and `Choupette` is an inhabitant of `Cat`, formally

```
(: Choupette Cat)
```

In this example `(: CA (<: Cat Animal))` would be an axiom, thus easily proven, but in general it may be necessary to construct a proof. For instance, it may not be directly known that `Cat` is a subtype of `Animal`, but instead it may be known that `Cat` is a subtype of `Mammal`, itself being a subtype of `Animal`. In this case, a proof of `(<: Cat Animal)` may look like

```
(: (STTrans CM MA) (<: Cat Animal))
```

where `STTrans` is the transitivity rule provided earlier, applied to axioms

```
(: CM (<: Cat Mammal))
```

```
(: MA (<: Mammal Animal))
```

Thus, the `nbrLimbs` call over `Choupette` may look like

```
(nbrLimbs (coerce (STTrans CM MA) Choupette))
```

With a formal specification of the subtyping relationship `<:` in hand, we can now define an ontology for the AI-DSL that the backward chainer can use as needed to reason about subtyping and type coercion while synthesizing AI service compositions. Please find below the miniature ontology mentioned earlier

```
;; Language
(: NS (<: NaturalLanguage String))

;; Text
(: TS (<: Text String))
(: TIT (<: (TextIn $l) Text))

;; URL
(: US (<: UniformResourceLocator String))
(: UTU (<: (UniformResourceLocatorOfType $t) UniformResourceLocator))

;; MIDI
(: MB (<: MusicalInstrumentDigitalInterface Bytes))

;; Audio
(: AB (<: Audio Bytes))
(: IA (<: Instrumental Audio))
(: VA (<: Vocals Audio))
(: VIV (<: (VocalsIn $l) Vocals))
(: SIA (<: (SongIn $l) Audio))
```

A graphical representation of the ontology is given in Figure 3.1 (Appendix A contains information of how it has been rendered).

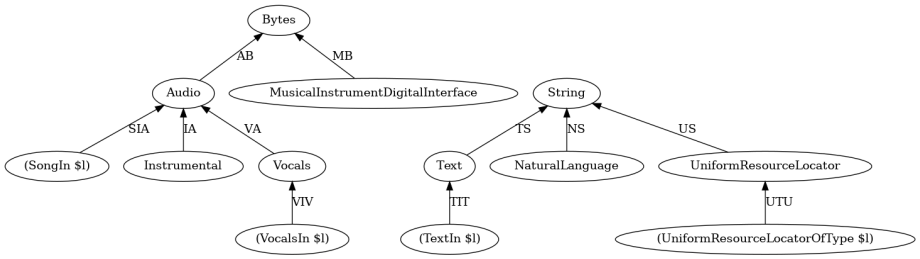


Figure 3.1: Graphical representation of the miniature ontology mentioned earlier. The nodes represent the types. The edges represent the subtyping relationships, for instance Audio is subtype of Bytes is represented by an arrow pointing to Bytes. The labels on the edges correspond to the names of axioms encoding the subtyping relationships.

One may observe that in this ontology all types directly or indirectly derive from primary types such as `String` and `Bytes`. These primary types comes Protobuf. What this means is that, if for instance an AI service outputs data of type `Text` to be received by a service that inputs data of type `String`, the type

coercion may not need to take place at run-time. Type coercion is, most of the time, only here to make sure the AI service composition is consistent with its AI service specifications. One may also observe that some types are parameterized, such as `(TextIn $l)`, which represents a text in a certain language `$l`. Having parameterized types in the ontology will especially turn out to be useful while reasoning about dependent types.

Given this ontology we can replace the Protobuf types by semantically richer types. Let us consider for instance the `naint.machine-translation.MkInput` constructor of `naint.machine-translation.Input` mentioned in Section 3.2 and recalled below

```
;; Define naint.machine-translation.Input constructor
(: naint.machine-translation.MkInput
   (-> String ; source_lang
       (-> String ; target_lang
           (-> String ; sentences_url
               naint.machine-translation.Input))))
```

One can replace the first two occurrences of `String` by `NaturalLanguage` and the last occurrence of `String` by `(UniformResourceLocatorOfType $t)` or even better by `(UniformResourceLocatorOfType Text)` since we know that this service translates text to text. But we can do even better than that and replace it by `(UniformResourceLocatorOfType (TextIn $l))` since we are given the source language as first argument. The question is how to specify the source language, for now being just represented as free variable, `$l`. This is where dependent types shine. To see how let us rewrite the entire constructor as follows

```
;; Define naint.machine-translation.Input constructor with
;; dependent types
(: naint.machine-translation.MkInput
   (-> (: $src-lang NaturalLanguage)
       (-> (: $dst-lang NaturalLanguage)
           (-> (UniformResourceLocatorOfType (TextIn $src-lang))
               (naint.machine-translation.Input $src-lang
                                                $dst-lang)))))
```

The variable `$l` has been replaced by `$src-lang`, the inhabitant of the first argument of `naint.machine-translation.MkInput`, the source language of the translation, et voila! It allows us to express that the third argument is a type that *depends* on the value of the first argument. Also, it goes without saying that replacing `String` by a type denoting a location to a datum, a URL, as opposed to the datum itself allows to catch at type-checking-time an error that would likely have otherwise occurred at run-time. Indeed, it is easy to make the mistake of plugging in input of the translator a service that directly outputs a text as opposed to a URL pointing to the text to translate. Finally, one should note that the `naint.machine-translation.Input` type has been replaced by a parameterized type, taking the source and destination languages as parameters, which is needed not to lose that information

downstream. Likewise the `naint.machine-translation.Output` type has been replaced by a parameterized type simply taking the destination language as parameter as the information about the source language is no longer needed. The `naint.machine-translation.translate` method thus looks like

```
;; Define naint.machine-translation.translate service method
(: naint.machine-translation.translate
   (-> (naint.machine-translation.Input $src-lang $dst-lang)
       (naint.machine-translation.Output $dst-lang)))
```

In the AI service composition experiments described in Chapter 4, the above has been further simplified by embedding the input and output constructors directly in the arguments of the `naint.machine-translation.translate` method, resulting into the following type signature

```
;; Define naint.machine-translation.translate service method
(: naint.machine-translation.translate
   (-> (: $src-lang NaturalLanguage)
       (: $dst-lang NaturalLanguage)
       (-> (: $url (UniformResourceLocatorOfType (TextIn $src-lang)))
           (TextIn $dst-lang))))
```

One may also notice that the fully curried representation has been replaced by some hybrid curried-uncurried one. Indeed the first two arguments, the source and destination languages, are uncurried while the last one, the URL pointing to the text to translate, is curried. This is because we have found that it mitigate the combinatorial explosion to cleaning separate parameters (like the source and destination languages) and data to be processed (like the text to be translated).

It all looks good and well, but there is one elephant in the Section. How to automate such type enrichment process? How to go from a given Protobuf specification to a MeTTa specification? How to choose the right semantically rich type in the ontology as a substitute for a semantically poor type in the Protobuf specification. And finally, how to come up with the ontology in the first place? As valid as these questions are, they are left aside in this technical report, but will undoubtedly resurface in the future when their time is right.

Chapter 4

Experimenting with AI Service Composition in MeTTa

We have written a few prototypes in MeTTa, all based on the backward chainers presented in Chapter 2, to synthesize AI service compositions. In this chapter we will describe the various experiments we have conducted with some benchmarks and a few lessons that we have learned along the way.

4.1 Experimental Setup

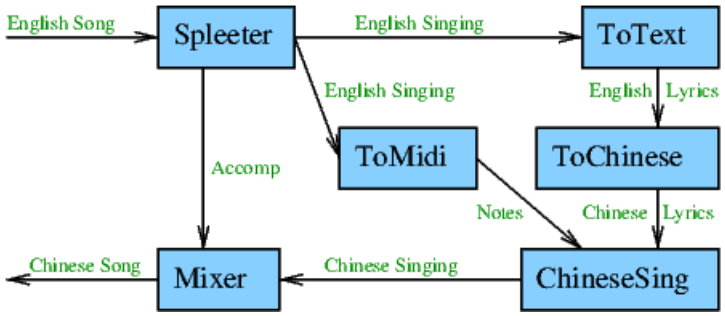
4.1.1 AI Service Compositions

After perusing the SingularityNET marketplace to find meaningful AI service compositions to attempt to synthesize them, we have retained two compositions:

1. **Speech Emotion Recognition**, obtained by sequentially combining two services, `speech2text-en` and `text-emotions`, provided by the NAINT organization, represented by the following flowchart-like graph:

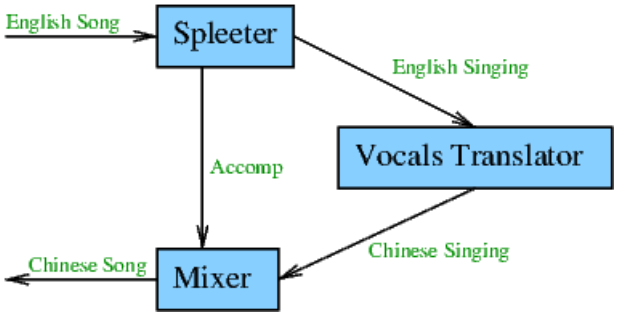
   The graph says it all, speech is turned into text, then emotion is detected from this text. One could easily envision a variation involving a parallel composition of another service that can detect emotion in speech directly, more focused and voice expression than semantics, but in this report we purposely leave it at that, a simple composition involving two services.

2. **English to Chinese Song Translation**, obtained to by combining the following services, `sound-spleeter` and `speech-recognition` from SingularityNET, `machine-translation` and `midi2voice-zh` from NAINT. In addition we have added an extra service, `tomidi`, to convert audio to MIDI. This service is not present on the SingularityNET marketplace so we had to handwrite it in MeTTa rather than obtain it from the crawler described in Chapter 3. We also had to add a micro service to mix two audio signals, `mixer`, also not present in the SingularityNET marketplace. The flowchart-like graph below represents the composition we wish to synthesize.



The flow of information is a bit more complicated here. An audio signal encoding an English song comes in, it is split into two audio signals, one for the instrumental and the other one for the vocals. The instrumental, `Accomp` in the graph, goes straight to the mixer. The vocals on the other hand is further duplicated into two signals, one converted into text, the lyrics, then translated to Chinese, the other converted into MIDI, the melody. Both the Chinese lyrics and the melody then join into the Chinese singing service to produce Chinese vocals. Finally the instrumental and the Chinese vocals get mixed to produce the Chinese song. Making this work in practice is actually more involved than just plugging these services as described and requires syncing the Chinese vocals and the instrumental. But regardless of the run-time result, what is of interest to us here is whether the AI-DSL is able to synthesize such AI service composition.

3. Additionally, our experiments contains a third AI service composition which is a scaled down version of the English to Chinese Song Translation shown below, defined to create an intermediary level of difficulty in synthesis between the Speech Emotion Recognition composition and the English to Chinese Song Translation composition.



In this composition the translation from English text to Chinese text, melody recognition and Chinese singing generation has been replaced by a single service, **Vocals Translator**. This service does not exists on the SingularityNET marketplace thus was handcoded in MeTTa specifically for the purpose of creating this down scaled variation.

4.1.2 Search Space

For each AI service composition, the search space is limited to the services involved. Meaning we do not dump the entire SingularityNET marketplace into the knowledge based of the chainer. So for instance for the Speech Emotion Recognition composition, only two services are described. This involves more than two axioms because services have data constructors and access functions. But this nonetheless dramatically reduces the search space. As we will see though, even with that simplification, in the absence of any pruning, the search can already easily become intractable. Much of the lessons learned during these experiments have to do with pruning the search.

4.1.3 Lambda Abstraction versus Combinators

There are at least two ways of describing AI service compositions.

- Lambda abstraction, where variables are used to capture and passed data to any part of the composition. Lambda abstraction is very powerful to express arbitrarily complex information flow between services but can somewhat obfuscate the structure of the composition.
- Combinatory logic is also equally powerful to describe arbitrarily complex information flow. The advantage of using combinators over lambda abstraction though is that the structure of the composition *can* be made more explicit. I insist on "can" because, as soon as combinators apply to higher order functions, things can become very confusing. Nonetheless, this way to describing the information flow can indeed represent more clearly what is the wiring between services. This way of describing programs by combining functions instead of carrying variables around is also called *tacit programming* [**?**].

There would be a third way to describe AI service compositions, using process algebras like the *pi*-calculus [**?**] or the *rho*-calculus [**?**]. We believe it is a promising future avenue of research but we have not explored it in this report.

4.1.4 Pruning Techniques

In order to prune the search we have experimented with the following:

- Limiting the flow of information to data, as opposed to functions. Meaning that on the wires on the composition only data can transit, not functions.
- Adding reduction rules to avoid synthesizing syntactically different yet semantically identical candidates. This is similar to how MOSES [**?**] uses reduction as pruning technique.
- Uncurrify services and combinators. This has the interesting effect of pruning the search by avoiding combining the services in weird, higher order ways, quite similar to limiting the information flow to data, but even stronger.

In the end we have found that using a combination of reduction and some well selected uncurried combinators leads to efficient synthesis of AI service compositions, in our limited setting anyway.

4.1.5 Bluebird and Phoenix Combinators

The well selected combinators aforementioned are:

- The *bluebird*, which is just another name for the regular function composition operator, that could be understood as simulating sequential composition.
- The *phoenix* is a less known yet powerful combinator that could be understood as simulating parallel composition [1].

The definitions of the bluebird and phoenix combinators are respectively provided in Tables 4.1 and 4.2.

[1] I am careful with my words because the λ-calculus is inadequate to model the difference between sequential and parallel computations. One would be better served using a process algebra, which is intended to be explored is the future.

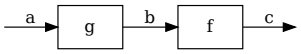
Name	bluebird
Symbol	.
Description	sequential composition
Type (Haskell)	`(b -> c) -> (a -> b) -> a -> c`
Type (MeTTa)	`(-> (-> $b $c) (-> $a $b) (-> $a $c))`
Definition	. f g x = f (g x)
Diagram	

Table 4.1: Bluebird combinator. A signal of type `a` get processed by `g` which outputs a signal of type `b`, itself then being processed by `f` which outputs a signal of type `c`.

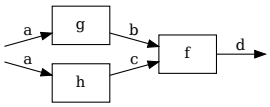
Name	phoenix
Symbol	Φ
Description	parallel composition (split then join)
Type (Haskell)	`(b -> c -> d) -> (a -> b) -> (a -> c) -> a -> d`
Type (MeTTa)	`(-> (-> $b $c $d) (-> $a $b) (-> $a $c) (-> $a $d))`
Definition	Φ f g h x = f (g x) (h x)
Diagram	

Table 4.2: Phoenix combinator. A signal of type `a` splits and get processed by `g` and `h` in parallel, which output signals of types `b` and `c` respectively, then rejoining to `f` which outputs a signal of type `d`.

It is reasonable to assume that these two combinators should be enough to describe any information flow between AI services as long as they do not require recursion [2].

4.2 Benchmarks

Experiments on the three AI service compositions presented above have been conducted on a AMD Ryzen 7 PRO 4750G with 64GB of RAM. The hyperon-experimental [**?**] MeTTa back-end was primarily used and all results reported in

[2] Assuming the identity combinator is also provided one may in fact only need the phoenix as it can be turned into the bluebird when combined with the identity, but that would not result to compositions as compact as having both the bluebird and the phoenix.

tables are for it. The MeTTaLog [**?**] back-end was also used but due to not being able to run on all experiments was not selected as the reference of comparison. As a result results with MeTTaLog are not systematically reported but given here and there just to provide an idea of its potential speed-up.

4.2.1 Speech Emotion Recognition

Experiments begin with the simplest AI service composition, **recognizing emotion in speech** described in Section 4.1.1, merely involving two services to be sequentially combined. In these experiments no ontology is used, instead the Protobuf types are directly used to specify the input and output of the desired composition. The query submitted to the backward chainer in all cases is

```
(: $prg (-> speech2text-en.Audio text-emotions.Output))
```

The results are detailed in table 4.3. The first three experiments, $\texttt{xp}_1$, $\texttt{xp}_2$ and $\texttt{xp}_3$, involve lambda abstraction and deliver poor (over 19 minutes of run-time in the best case). The poor performances are due to lambda abstraction requiring more depth to describe the desired composition, thus dramatically increasing the search space. It is also due to the inherent expressive power of lambda abstraction, in particular its capacity to build higher order functions, enabling far more combinations than necessary. When higher order lambda abstraction is disabled (experiment $\texttt{xp}_2$), meaning that the variables created by lambda abstraction can only bind to data instead of functions, the time is considerably reduced, but is still large (23m40s). Pruning the search space by applying reduction rules (experiment $\texttt{xp}_3$) reduces run-time a bit further to 19m6s. Notably the number of resulting candidates is lower as well, going from 44 to 20. The reduction rules used are simply the following two

```
;; Identity application reduction
(= ((\ (: $x $t) $x) $y) $y))
;; Constant function application reduction
(= ((\ (: (s z) $t) z) $x) z)
```

It is good because less candidates means less work to select the final composition by the downstream process. When using MeTTaLog, results for $\texttt{xp}_3$ goes down to 1m21s, which is far better than 19m6s with hyperon-experimental, but still unacceptable for such simple AI service composition. The best solution synthesized by experiments $\texttt{xp}_1$, $\texttt{xp}_2$ and $\texttt{xp}_3$ is given below.

```
(: (\ (: z speech2text-en.Audio)
      (text-emotions.recognize
       (text-emotions.MkInput
        (speech2text-en.Text.text
         (speech2text-en.s2t z)))))
   (-> speech2text-en.Audio text-emotions.Output))
```

Note that other semantically equivalent but more complex solutions, or even incorrect ones, where also provided among the results. An example of a correct but more complex solution is

```
(: (\ (: z speech2text-en.Audio)
      ((\ (: (s z) String)
          (text-emotions.recognize (text-emotions.MkInput (s z))))
        ((\ (: (s z) speech2text-en.Text)
            (speech2text-en.Text.text (s z)))
          ((\ (: (s z) speech2text-en.Text)
              (s z))
            (speech2text-en.s2t z)))))
   (-> speech2text-en.Audio text-emotions.Output))
```

An example of a simpler but incorrect solution is

```
(: (\ (: z speech2text-en.Audio)
      (text-emotions.MkOutput
       (speech2text-en.Text.text
        (speech2text-en.s2t z))))
  (-> speech2text-en.Audio text-emotions.Output))
```

It is incorrect because it never invokes the `text-emotions.recognize` service. Instead it tricks the type by directly constructing an emotion from the output of `speech2text-en.s2t` using the `text-emotions.MkOutput` constructor. This is a good example of an under-specified formal description. Obviously, `text-emotions.MkOutput` should not be allowed to construct an emotion from an arbitrary text. This is where the ontology comes into play and will be explored in Section 4.2.2.

Using the bluebird and phoenix combinators instead of lambda abstraction (experiment $\texttt{xp}_4$) considerably improve performances, going from 19m6s to 6.86s. This is explained by the fact that now only a depth of 4 is required to synthesize the desired composition. The bluebird and the phoenix combinators seem to strike the right balance between expressiveness and restrictiveness. In other words, they seem to be placed at the right level of abstraction to describe AI service compositions, compared to say a more expressive and foundational combinator sets such as I, S and K. Also, it is not presented in the report but one of our first experiments involved using the I, S, K combinators. And if I recall correctly it was performance-wise even worse than lambda abstraction. The solution that was synthesized by experiment $\texttt{xp}_4$ is given below.

```
(: ((. text-emotions.recognize)
    ((. text-emotions.MkInput)
      ((. speech2text-en.Text.text)
        speech2text-en.s2t)))
  (-> speech2text-en.Audio text-emotions.Output))
```

Obviously, only the bluebird operator appears in the solution since it is entirely built of sequential compositions.

Another considerable speed-up is gained when using uncurried versions of the bluebird and phoenix combinators (experiment $\texttt{xp}_5$) going from 6.86s to 0.69s. This allows to represent the AI service composition even more compactly, requiring only a depth of 2. Additionally, using uncurried version makes it impossible for the combinator to create higher order functions, thus pruning the search space even further by only considering composition such that data, but not functions, transit between services. Here enabling reduction (experiment $\texttt{xp}_6$) has no impact on run-time. However, it reduces the number of candidates from 3 to 2. The best solution that was synthesized by experiments $\texttt{xp}_5$ and $\texttt{xp}_6$ is given below.

```
(: (. text-emotions.recognize
      (. text-emotions.MkInput
         (. speech2text-en.Text.text
            speech2text-en.s2t)))
  (-> speech2text-en.Audio text-emotions.Output))
```

Experiment ID	`xp`$_1$
File name	`rse-lambda-xp.metta`
Description	The composition is achieved using lambda abstraction.
Parameters	Reduct: False, OnlyData: False, Depth: 5
Results	TODO
Experiment ID	`xp`$_2$
File name	`rse-lambda-xp.metta`
Description	Like `xp`$_1$ but the variables in the lambda abstraction can only bind to data, not functions.
Parameters	Reduct: False, OnlyData: True, Depth: 5
Results	Time: 23m40s, NumCandidates: 44
Experiment ID	`xp`$_3$
File name	`rse-lambda-xp.metta`
Description	Like `xp`$_2$ but reduction is enabled.
Parameters	Reduct: True, OnlyData: True, Depth: 5
Results	Time: 19m6s, NumCandidates: 20
Experiment ID	`xp`$_4$
File name	`rse-combinator-xp.metta`
Description	The composition is achieved using the bluebird combinator.
Parameters	Reduct: False, Uncurried: False, Depth: 4
Results	Time: 6.86s, NumCandidates: 30
Experiment ID	`xp`$_5$
File name	`rse-combinator-curried-xp.metta`
Description	Like `xp`$_4$ but the bluebird operator is uncurried to further constrain its applicability.
Parameters	Reduct: False, Uncurried: True, Depth: 2
Results	Time: 0.69s, NumCandidates: 3
Experiment ID	`xp`$_6$
File name	`rse-combinator-curried-xp.metta`
Description	Like `xp`$_5$ but reduction is enabled.
Parameters	Reduct: True, Uncurried: True, Depth: 2
Results	Time: 0.68s, NumCandidates: 2

Table 4.3: AI service composition results for synthesizing AI service compositions for recognizing emotion in speech. The files involved in reproducing the experiments can be found under the folder `experimental/ai-service-composition/recognize-speech-emotion/` from the root folder of the AI-DSL repository [**?**].

4.2.2 English To Chinese Song

We experiment with two versions of the English To Chinese Song AI service composition, scaled down and normal sizes, as described in Section 4.1.1. Like above we experiments with lambda abstraction, bluebird and phoenix combinators, reduction and uncurried combinators. Additionally we use the ontology described in Section 3.3 specifically tailored for this AI service composition. In

all cases the query submitted to the backward chainer is

```
(: $prg (-> (: $x (SongIn "English")) (SongIn "Chinese")))
```

Meaning that we want to synthesize the program that takes in input a song in English and outputs a song in Chinese. The query does not specify more information than that. For instance, it does not specify that the song in Chinese must be the same as the English one, except the singing part. As such it still constitutes an over simplification of what we want to achieve. More complex AI service composition formal specifications will be explored in the future. However the size of the AI service composition (the normal one, not the scaled down one) resembles what we want achieve. Indeed, if we consider all services plus the necessary glue to connect them such as combinators, microservices and type coercion to handle the ontology, we are already talking about combining 14 unique components. Which I believe is above the size we would expect such synthesis process to start providing real world value. Of course one the reasons we are able to do it is because we already know a priori which components are going to be used among a much larger set, constituted in large part from the AI services available on the SingularityNET marketplace.

Table 4.4 contains the results of the experiments.

Experiment ID	$\texttt{xp}_7$
File name	`etcs-dtl-atw-ontology-syn-sd-xp.metta`
Description	Scaled down English to Chinese song AI service composition. The composition is achieved using lambda abstraction.
Parameters	Reduct: True, OnlyData: True, Depth: 6
Results	TODO
Experiment ID	$\texttt{xp}_8$
File name	`etcs-combinator-sd-xp.metta`
Description	Like $\texttt{xp}_7$ but lambda abstraction is replaced by bluebird and phoenix combinators.
Parameters	Reduct: False, Uncurried: False, Depth: 5
Results	Time: 1m42s, NumCandidates: 29
Experiment ID	$\texttt{xp}_9$
File name	`etcs-combinator-sd-xp.metta`
Description	Like $\texttt{xp}_8$ but reduction is enabled.
Parameters	Reduct: True, Uncurried: False, Depth: 5
Results	Time: 42.7s, NumCandidates: 1
Experiment ID	$\texttt{xp}_{10}$
File name	`etcs-combinator-sd-data-uncurried-xp.metta`
Description	Like $\texttt{xp}_9$ but the bluebird and phoenix combinators are uncurried.
Parameters	Reduct: True, Uncurried: True, Depth: 3
Results	Time: 1.6s, NumCandidates: 1
Experiment ID	$\texttt{xp}_{11}$
File name	`etcs-combinator-data-uncurry-xp.metta`
Description	Like $\texttt{xp}_{10}$ but on the full English to Chinese song AI service composition. Also, reduction is disabled.
Parameters	Reduct: False, Uncurried: True, Depth: 6
Results	Time: ?, NumCandidates: ?
Experiment ID	$\texttt{xp}_{12}$
File name	`etcs-combinator-data-uncurry-xp.metta`
Description	Like $\texttt{xp}_{11}$ but reduction is enabled.
Parameters	Reduct: True, Uncurried: True, Depth: 6
Results	Time: 7m56s, NumCandidates: 18

Table 4.4: AI service composition results for synthesizing AI service compositions to turn an English into a Chinese song. The files involved in reproducing the experiments can be found under the folder `experimental/ai-service-composition/english-to-chinese-song/` from the root folder of the AI-DSL repository [**?**].

Experiment $\texttt{xp}_7$ uses lambda abstraction, but enables reduction and binding only data to variables of lambda abstraction right away. In spite of that it is NEXT. The next experiment, $\texttt{xp}_8$, moves on to bluebird and phoenix combinators. Even without reduction, and still using curried combinators, the best

solution is found in 1m42s. Enabling reduction in experiment $\texttt{xp}_9$ brings that down to 42.7s. NEXT: This number may seem too big but in MeTTaLog it takes less than a second. Moreover the number of candidates goes from 29 to 1, only retaining the best solution. That is because we have introduced more reduction rules, 6 in total. In both experiments the best solution is

```
(: ((. (((Φ mix) accomp) ((. translate) vocals))) spleeter)
   (-> (: $x (SongIn "English")) (SongIn "Chinese")))
```

Experiment $\texttt{xp}_{10}$ is like $\texttt{xp}_9$ but uses uncurried bluebird and phoenix combinators, bringing synthesis time down to 1.6s, which is not only faster but produces a candidate that is easier to read

```
(: (. (Φ mix accomp (. translate vocals)) spleeter)
   (-> (: $x (SongIn "English")) (SongIn "Chinese")))
```

One the goals of the AI-DSL is to come up with a user friendly DSL to describe AI service compositions. It seems that combining tacit programming, thus relieving us from having to use intermediary variables to represent the flow of information, with the bluebird and phoenix combinators produces the elegance we are looking for! We are now ready to synthesize the full English to Chinese song AI service composition. Because the full composition is much larger than what has been previously attempted we skip lambda abstraction and curried combinators. Instead we run two experiments using uncurried bluebird and phoenix combinators, one with reduction disabled (experiment $\texttt{xp}_{11}$) and the other with reduction enabled (experiment $\texttt{xp}_{12}$). In both experiments the best candidate is

```
(: (. (Φ mixer.mix
         sound-spleeter.DTLOutput.accomp
         (. (Φ midi2voice-zh.singingZH
               (. (machine-translation.translate "English" "Chinese")
                  (. upload speech-recognition.s2t))
               tomidi.a2m)
            (. (coerce IA) sound-spleeter.DTLOutput.accomp)))
      sound-spleeter.spleeter)
   (-> (: $x (SongIn "English")) (SongIn "Chinese")))
```

Figure 4.1 shows a graphical representation of this candidate produced by the MeTTa to DOT converter described in Appendix A.

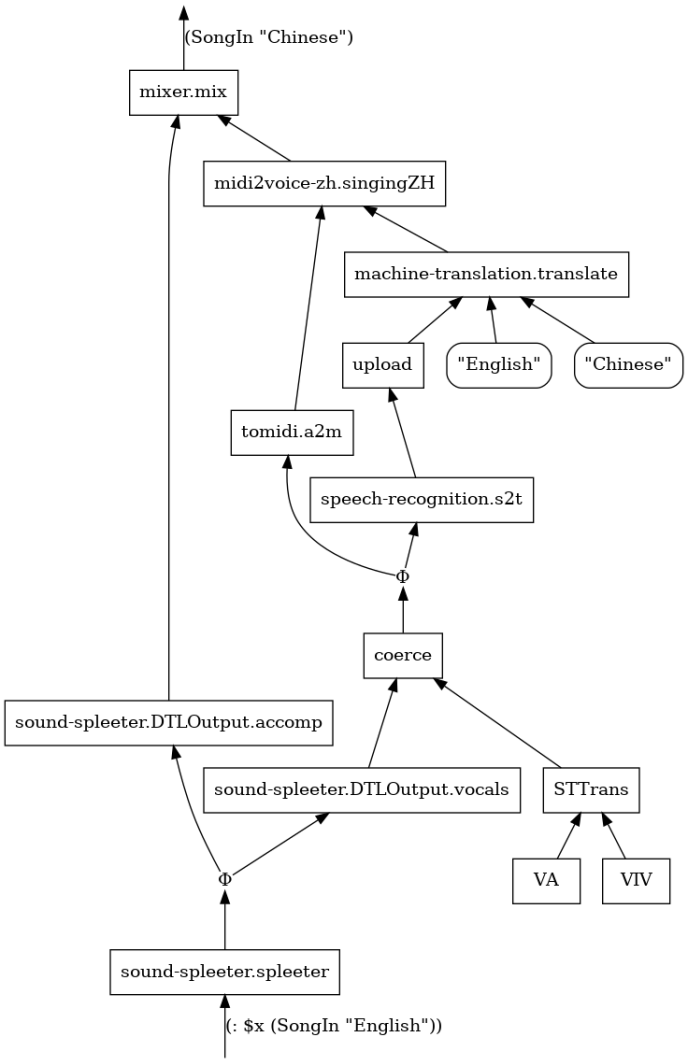


Figure 4.1: Graphical representation of the best candidate of experiments $\texttt{xp}_{11}$ and $\texttt{xp}_{12}$. The nodes represent services like `tomidi.a2m`, microservices like `upload`, type coercion like `coerce`, subtyping proof constructors used by type coercion like `SSTrans`. The edges represent the information flow between the nodes. The labels on the input and output edges of the composition represents the input and output types respectively.

However $\texttt{xp}_{11}$ took XXX and produced XXX candidates, while $\texttt{xp}_{12}$ took only 7m56s and produced 18 candidates. Of these 18 candidates some are still

incorrect due to the under-specification of the English to Chinese song AI service composition, such as

```
(: (Φ mixer.mix
      (. sound-spleeter.DTLOutput.accomp sound-spleeter.spleeter)
      (Φ midi2voice-zh.singingZH
         (. (machine-translation.translate "English" "Chinese")
            (. upload (. speech-recognition.s2t (coerce SIA))))
         (. tomidi.a2m (coerce SIA))))
  (-> (: $x (SongIn "English")) (SongIn "Chinese")))
```

It is incorrect because XXX. Figure 4.2 provides a graphical representation of this incorrect candidate.

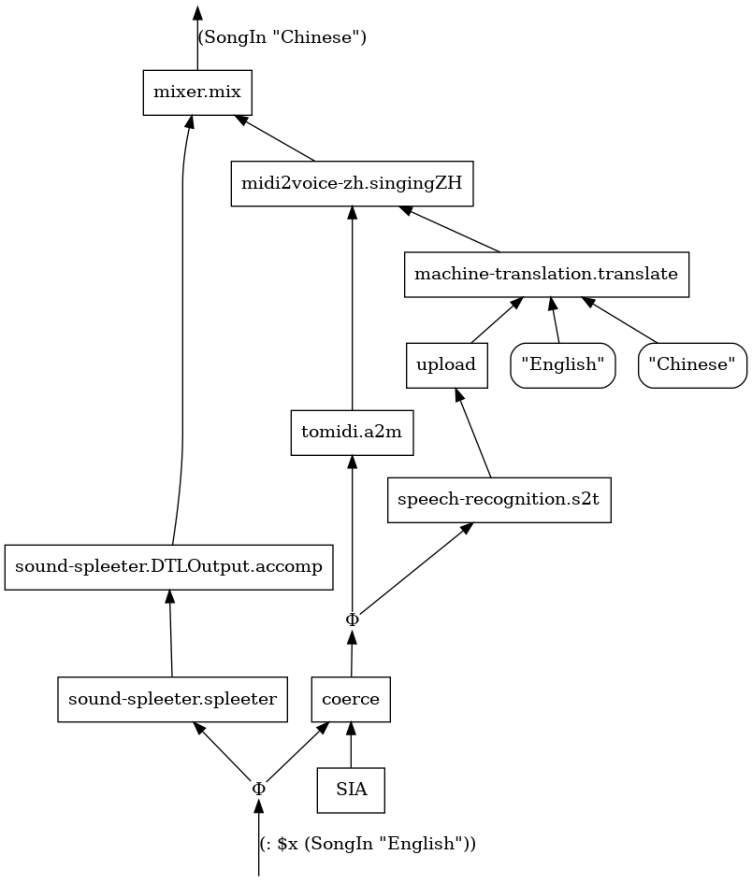


Figure 4.2: Graphical representation of an incorrect candidate.

Additionally see Figure 4.3 for a graphical representation of the best candi-

date with all edges annotated by types.

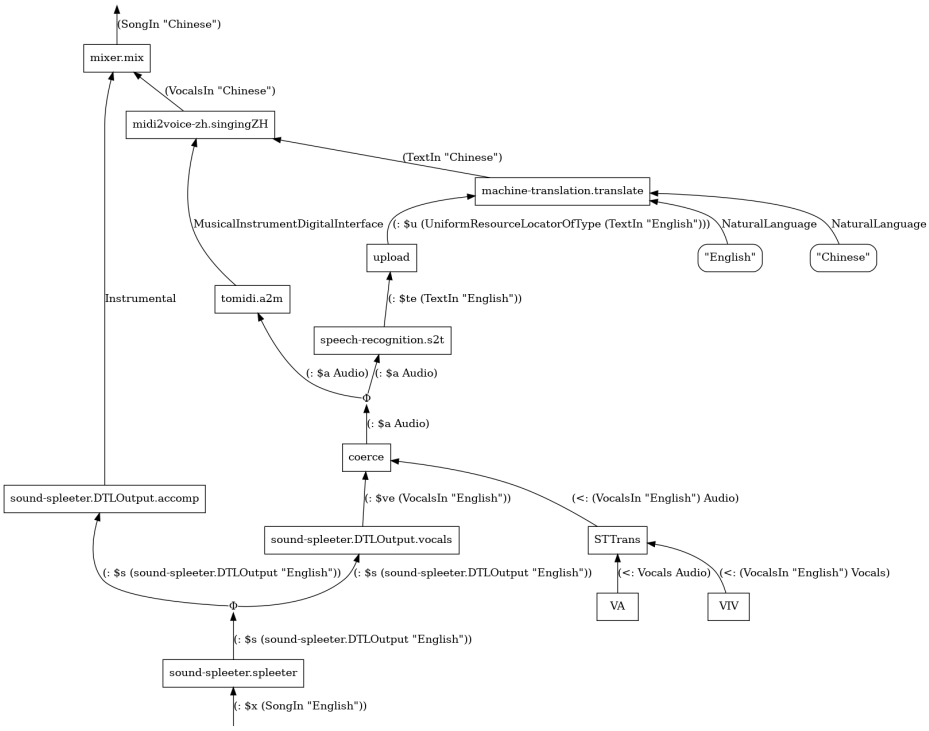


Figure 4.3: Graphical representation of the best candidates with all edges annotated by types. This allows to see the type of the data that transit between services.

NEXT: mention about partial currification of services (parameters

Chapter 5

Conclusion

5.1 Future Developments

1. Manage resources, financial, computational.
2. Reason about uncertainty of outcomes (probability of failing, average accuracy of the results, etc).
3. Use infer grained language that captures not just functional semantics but also run-time behavior.
4. How to build the ontology and enrich Protobuf specifications to leverage such ontology.

5.2 Acknowledgments

Thanks to Matt Ikle and Khellar Crawford for our numerous stimulating discussions. Thanks to Remy Clarke for mentioning APL and BQN which led me to learn about tacit programming. Thanks to Douglas Miles for making MeTTaLog which has been tremendously helpful for some experiments and getting to reach results at all in some circumstances. Thanks to Vitaly Bognanov for making the hyperon-experimental back-end of MeTTa which was used across all experiments to establish a fair comparison. Thanks to the SingularityNET Foundation for funding this work.

Appendix A

DOT Format Conversion

Bibliography

[1] jq. https://jqlang.org

[2] MeTTa Language Official Website. https://metta-lang.dev

[3] protobuf-metta. https://github.com/trueagi-io/protobuf-metta

[4] SingularityNET CLI. https://github.com/singnet/snet-cli

[5] Idris, Idris Homepage (2021), https://www.idris-lang.org/

[6] Geisweiller, N.: Chaining Repository. https://github.com/trueagi-io/chaining

[7] Geisweiller, N.: Curried Backward Chainer. https://github.com/trueagi-io/chaining/blob/main/experimental/curried-chaining/curried-chainer.metta

[8] Geisweiller, N.: Examples of singularitynet ai service assemblages (2023), https://github.com/singnet/ai-dsl/blob/master/doc/technical-reports/2022/snet-service-assemblages.pdf

[9] Nil Geisweiller, Matthew Ikle, D.D.S.R.: Ai-dsl technical report (may to septempber 2022) (2023), https://github.com/singnet/ai-dsl/blob/master/doc/technical-reports/2022/ai-dsl-techrep-2022-oct.pdf