Sofia University "St. Kliment Ohridski"
Faculty of Mathemathics and Informatics
Department of Mathematical Logic and Applications

Master's thesis

Combinatory Categorial Grammars for Geospatial Queries

Angel Pavlov Angelov

M.Sc. Program "Computational Linguistics"

Faculty number:	25744
Email:	`angelpa1@uni-sofia.bg`

Supervisor:	Assoc. Prof. Trifon Trifonov
Advisor:	Assist. Prof. Stefan Gerdzhikov

2019

Contents

1 Introduction

This project seeks to provide means for translating from a subset of English into Overpass queries using a Combinatory Categorial Grammar.

1.1 Motivation

The subject of this thesis came into being rather randomly. First, the following ingredients were implanted into a cadaver:

- My affection towards maps and locations
- My contempt towards ~~black boxes invented by aliens~~ machine learning
- My desire to do something with λ-calculus / functional languages / compiling
- My having read the wiki page about Overpass several days before thinking about a thesis subject

Then, said cadaver was strapped on a table in a[1] tower and its limbs connected to the tower's lightning rod. The lightning that figuratively struck the lightning rod came as a *sample subject* presented by my (as of now) supervisor for this thesis — something along the lines of "Translating a subset of a natural language into a formal language (SQL or Haskell) via Combinatory Categorial Grammar". After refracting the aforementioned ingredients through the plasmatic halo of this sample subject, this thesis was implemented, with no regard for any usefulness or state-of-the-art–ness. The assessment of such qualities is left as exercise to the reader.

1.2 Motivation

It is often the case that humans need to communicate a concrete question or order (a query) to a computer. The most straightforward way for this to happen is to have the human write out their query in a domain-specific language (DSL), specially crafted to handle the given type of query. This approach, albeit very efficient when done correctly, requires substantial upfront effort: the human needs to get familiar with the DSL and must have access to a specific intermediary (usually a keyboard and a screen). Thus, in some cases it is desirable to form an expression in a natural language (or a language very similar to a subset of a natural language) and have it translated to a query in the given DSL.

The point is not to translate *any* sentence in the source language that has semantics applicable to the target DSL, but rather to define a subset of the source language, in which a human could express a query reasonably easily.

Most DSLs have compositional semantics, which makes them great candidates for generation using Combinatory Category Grammars (CCGs) [BB11, p. 181].

[1]The tower of The Faculty of Chemistry and Pharmacy of St. Kliment Ohridski University of Sofia, located at 42.67455,23.33225.

One such DSL is the Overpass language, used by the open source project OpenStreetMap for making queries to the map database. It has been chosen as a concrete target language for this project in order to assess the usefulness of CCG for translating from a natural language subset to a compositional DSL.

1.3 The waving of hands

This section will informally give a basic idea for all concepts used throughout the text with the sole purpose to give some intuition about the project end-to-end.

Our goal is to have our user write a query like this:

```
pharmacies near parking spaces in Berlin
```

to get circles on a map as those shown in fig. 1.1.

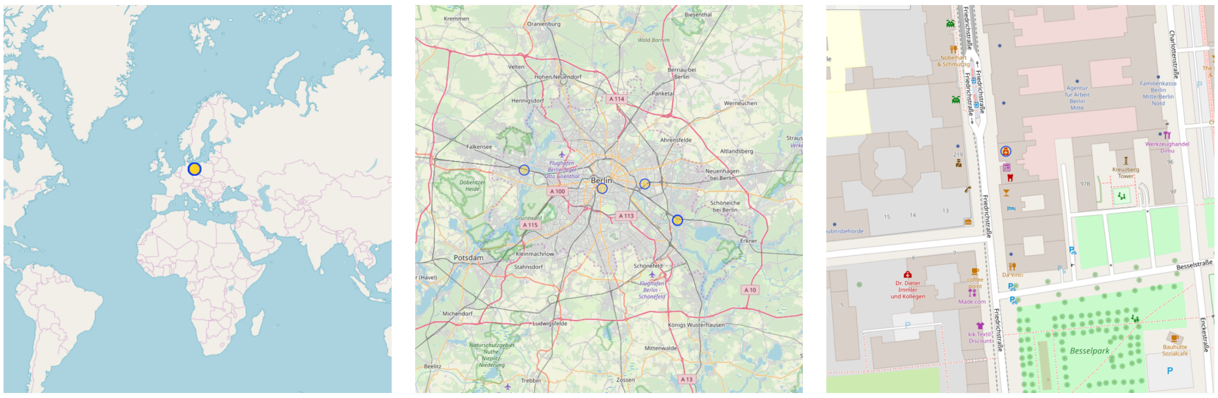


Figure 1.1: Desired result for the query "pharmacies near parking spaces in Berlin"

To do this, we will first construct a Combinatory Categorial Grammar which parses such queries. The central idea of CCG is that each word (or terminal) is assigned a set of *categories*, which work in the same way as types work in programming languages[2]. Semantically, each word returns something as a result and may take other things as arguments.

Here, our central type will be represented by the type symbol $GSet$ and will mean "a set of geographic objects". Each basic type is also a category.

So, we can immediately assign the category of `pharmacies` and of `Berlin` to be $GSet$ — they will have the semantics of "the set of all pharmacies" and "the singleton of the city of Berlin" respectively:

$$\begin{aligned} pharmacies &: GSet \\ Berlin &: GSet \end{aligned}$$

Now, we want `parking spaces` to be of the $GSet$ category and have the semantics "the set of all parking spaces". One way to do this is to assign a special category $Spaces$ to the

[2]Especially curried programming languages like Haskell

word `spaces`, and make `parking` take $Spaces$ as an argument on the right and return $GSet$. We will denote this "argument on the right" construction by a forward slash:

$$\begin{aligned} parking &: GSet/Spaces \\ spaces &: Spaces \end{aligned}$$

Similarly, arguments on the left will be denoted by a backward slash.

Now, we need to assign a category to the word `in`. We want to parse the construction "something in something", thus we want to take one $GSet$ on the left and one $GSet$ on the right, and return another $GSet$:

$$in : (GSet\backslash GSet)/GSet$$

This means that `in` will take a $GSet$ on the right and produce something which takes another $GSet$ on the left and returns the final $GSet$. In our case, `in Berlin` takes some $GSet$ on the left and returns a $GSet$. An alternative category that would also work is the following:

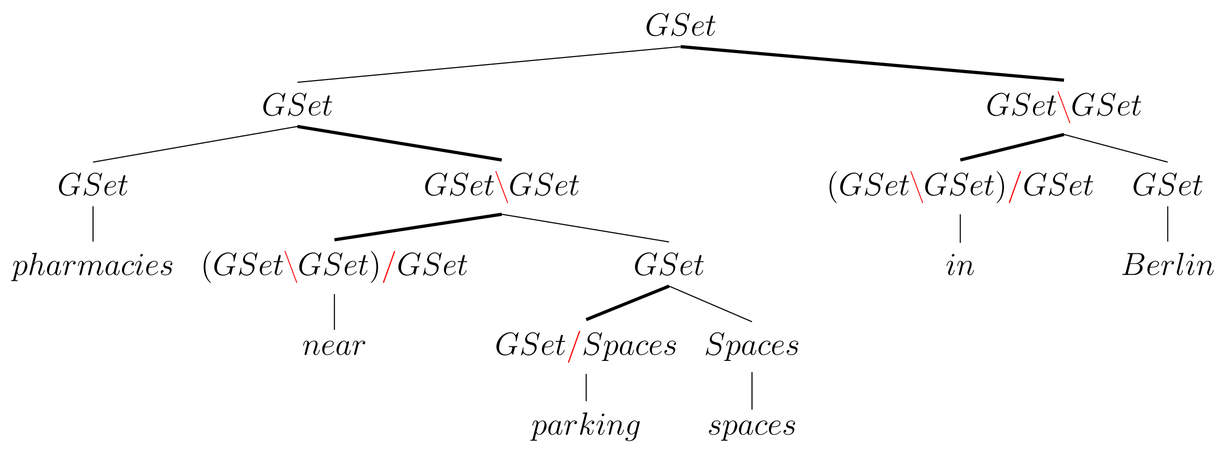
$$in : (GSet/GSet)\backslash GSet$$

In this case, `in Berlin` cannot be parsed while `parking spaces in` will return something that will gladly take `Berlin` on the right and produce a $GSet$.

The syntax of "near" is similar to the syntax of "in", so we will assign the same category. Finally, we obtain this simple grammar:

$$\begin{aligned} pharmacies &: GSet \\ Berlin &: GSet \\ parking &: GSet/Spaces \\ spaces &: Spaces \\ in &: (GSet\backslash GSet)/GSet \\ near &: (GSet\backslash GSet)/GSet \end{aligned}$$

This grammar will produce a valid parse for our query. The most convenient way to illustrate such parses is by means of a *derivation tree*:



Here, a thick line signifies the "function" while a thin line signifies the "argument".

We can see that this derivation tree simply takes the building blocks produced by the words and combines them in pairs until we get a single result. This means we have assumed that the semantics of the language are *compositional* — larger sentences have semantics that can be described by composing the semantics of its constituent subsentences.

Now, if we replace each word in the derivation tree by a building block of another target language, and assume that we have chosen blocks with the same semantics, we will compose a sentence in the other language that has the same meaning! Of course, those suppositions (the compositional nature of the language and that we can chose terminals with the same semantics) are utterly naïve, especially for natural languages. But, if we restrict ourselves to a limited domain within the natural language and generate sentences in a formal language, those assumptions might give us acceptable results. Ideally, we want to generate the following

Overpass query:

```
( node["amenity" = "parking_space"]; ) -> .x1;
( area["name" = "Berlin"]; area["int_name" = "Berlin"]; area["name:en"
   = "Berlin"]; ) -> .x2;
( node(area.x2)(around.x1:100.0)["amenity" = "pharmacy"]; ) -> .x3;
.x3 out;
```

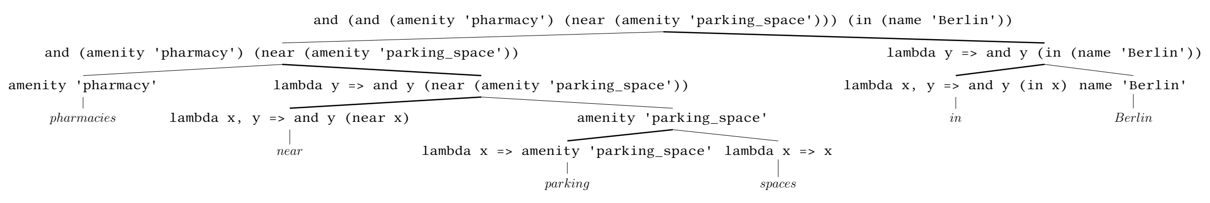
Since the compositional nature in Overpass is hidden, instead of generating it directly we will generate a query in a custom intermediate language we will call Minipass. In Minipass, the same query will look like this:

```
and (and (amenity 'pharmacy') (near (amenity 'parking_space'))) (in
    (name 'Berlin'))
```

We will now assign a Minipass term to each of the words in our example (specified after the @ symbol):

pharmacies	:	$GSet$	@ `amenity 'pharmacy'`
Berlin	:	$GSet$	@ `name 'Berlin'`
parking	:	$GSet/Spaces$	@ `lambda x => amenity 'parking_space'`
spaces	:	$Spaces$	@ `lambda x => x`
in	:	$(GSet\backslash GSet)/GSet$	@ `lambda x, y => and y (in x)`
near	:	$(GSet\backslash GSet)/GSet$	@ `lambda x, y => and y (near x)`

We can now see how the tree will compose these terms:


and (and (amenity 'pharmacy') (near (amenity 'parking_space'))) (in (name 'Berlin'))
and (amenity 'pharmacy') (near (amenity 'parking_space'))
lambda y => and y (in (name 'Berlin'))
amenity 'pharmacy'
lambda y => and y (near (amenity 'parking_space'))
lambda x, y => and y (in x)
name 'Berlin'
pharmacies
lambda x, y => and y (near x)
amenity 'parking_space'
in
Berlin
near
lambda x => amenity 'parking_space'
lambda x => x
parking
spaces

Note that "lambda" expressions have been "reduced" in order to decrease clutter. We will later formally explain their semantics and how to do that.

This produces the exact Minipass query we wanted. The grammar can now be extended to cover many more types of queries.

The last thing we will do is create a translator from Minipass to Overpass to complete the pipeline in fig. 1.2.

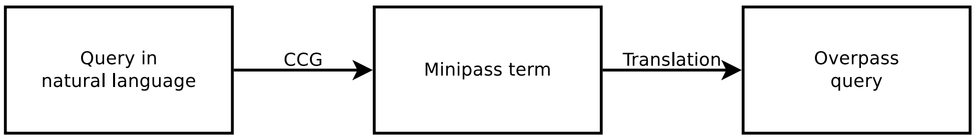


Figure 1.2: General pipeline for a query

1.4 Overview of the Overpass language

The Overpass language [wik19a] has been created by the OpenStreetMap community in order to be able to make more complex queries to map data.

Overpass has two different query fronteds, which means that the same AST may be built in two different ways: a *XML*-based frontend and an *Overpass QL*–based frontend. The XML Overpass QL frontend has been better maintained, has more features and is easier to generate programmatically — thus, it was chosen as a target language for this project. Whenever we talk about "the Overpass language", we shall refer to Overpass QL.

This section will present a subset of Overpass QL that is relevant to this project: for a more complete overview, refer to [wik19b].

1.4.1 Map data

Overpass lets us examine the universe of OpenStreetMap map data. It consists of *objects*, where each object has a set of key-value tags (keys and values are strings but some values may encode a more complex structure within a string).

An object may be of one of the following types:

- *node* — a point on the map, identified by latitude, longitude, and a node ID. Nodes represent point-like features like trees, fountains, businesses, mountain peaks, etc.
- *relation* — a grouping of multiple objects by some common feature. Examples of relations are:
  - all stops in a bus route
  - all intersections of a highway
  - poles of a power grid line

  Relations may contain nodes, ways and other relations. Each member of a relation may have an optional *role*, which is a text field that may be used to further group relation elements.
- *way* — an ordered list of nodes which is used for defining continuous features. Ways may be open or closed (closed ways have an identical first and last node). Open ways are used for defining things like roads, rivers, and railways, while closed ways are used for defining polygons (building boundaries, area borders, closed features).

- *area* — not an actual OpenStreetMap object. Closed ways or relations which contain closed ways may also be classified as an area by some criteria (one of which is the presence of an `area=yes` tag). Areas allow additional operations to the ones usable only for ways taking advantage of a cached optimised representation of each polygon.

1.4.2 Basic language structure

The abstraction Overpass uses to represent map data is called a *set*, where a set can contain any number of objects. Sets are heterogeneous and thus there is no limit on the variety of objects that may reside in a single set.

OverpassQL consists of a list of *statements* which retrieve, manipulate, or output such sets. Sets may be stored in variables: the syntax for referring to a variable is a dot followed by alphanumeric characters. Statements are terminated by semicolons. Each statement has an *output set* and may have input sets. When unspecified, an input or output set is implicitly set to the *default set* (`._`).

There is one special statement — the "`out`" statement, which outputs all objects from an input set to the user in some form. The output format may be set via specific meta-statements or by API options — this, however, is not within scope. In the Overpass Turbo web interface, results may be visualised as dots and polygons on a map. The syntax of the "`out`" statement is:

```
out <input set>;
```

Example 1.1. The following statement saves all nodes with name Foo within the set variable `._`:

```
node[name="Foo"];
out;
```

It is equivalent to the following statement:

```
node[name="Foo"] -> ._;
out ._;
```

In order to use `foo` as an output set, we could do:

```
node[name="Foo"] -> .foo;
out .foo;
```

The most basic group of statements are *retrieval statements*. They may retrieve specific type of objects depending on what *object keyword* was used. Object keywords are the following:

`node`	retrieves nodes
`way`	retrieves ways
`relation`	retrieves relations
`area`	retrieves areas
`nwr`	retrieves nodes, ways, and relations

To form a retrieval statement, an object keyword is followed by one or more *filters*:

```
<object keyword><filter><filter>...<filter>
```

Objects matching *all* filters are stored in the output set.

1.4.3 Tag filters

The most basic group of statements retrieve objects depending on whether a specific filter matches its tags. All tag filters are specified within square braces.

These are the possible tag filters[3]:

Filter	Negation	Effect
`["tag"]`	`[!"tag"]`	Matches if the object has a tag with key `tag`
`["tag"="content"]`	`["tag"!="content"]`	Matches if the object has a tag with key `tag` and value `content`
`["tag"~"regex"]`	`["tag"!~"regex"]`	Matches if the object has a tag with key `tag` and value that matches `regex`
`["tag"~"regex",i]`	`["tag"!~"regex",i]`	Matches if the object has a tag with key `tag` and value that matches `regex`, ignoring case
`[~"regex1"~"regex2"]`	`[~"regex1"!~"regex2"]`	Matches if the object has a tag with key that matches `regex1` and value that matches `regex2`

[3]All double quotes within filters may be replaced by single quotes (in order to escape double quotes within the value) or omitted (when the value contains only letters)

Example 1.2. Get nodes with amenity "cafe":

```
node[amenity="cafe"];
out;
```

Get nodes with amenity "cafe" and name "Moondeers":

```
node[amenity="cafe"][name="Moondeers"];
out;
```

Get nodes with amenity "cafe" and name "moondeers", case insensitive:

```
node[amenity="cafe"][name~"^moondeers$",i];
out;
```

Get areas with an "administrative" tag:

```
area["administrative"];
out;
```

1.4.4 Relative filters

Some filters can use an input set to influence the retrieval query.

- Set intersection: this filter only matches objects that are within the input set. The syntax is a dot followed by the input set's name.

  Example 1.3. Get nodes with amenity "cafe":

  ```
  node[amenity="cafe"] -> .cafes;
  node.cafes;
  ```

  Get nodes with amenity "cafe" and name "Moondeers":

  ```
  node[amenity="cafe"] -> .cafes;
  node.cafes[name="Moondeers"];
  ```

- Objects within an area: get objects that are physically inside of areas in the input set. The syntax uses the `area` keyword within round braces:

```
(area.<input set>)
```

Example 1.4. Nodes in Frankfurt:

```
area[name="Frankfurt"] -> fr;
node(area.fr);
out;
```

- Physical distance: get objects within said distance from objects in the input set. The syntax involves using the `around` keyword within round braces:

```
(around.<input set>:<radius in metres>)
```

Example 1.5. Cafes within 120m of a parking space:

```
node[amenity="parking_space"] -> pspaces;
node[amenity="cafe"](around.pspaces:120) -> x;
out x;
```

Cinemas within 100m of bus stops in Bonn:

```
area[name="Bonn"];
node(area)[highway=bus_stop];
node(around:100)[amenity=cinema];
out;
```

1.4.5 Set union

Several statements that produce an output set may be chained within round brackets. This is a *union* statement and produces the union of all output sets. Bare input sets (preceded by a dot) may also take part in a union.

Example 1.6. Nodes and areas with an "administrative" tag:

```
(node["administrative"]; area["administrative"];);
out;
```

Cafes in Frankfurt and Bonn:

```
node[amenity="cafe"] -> .cafes;
area[name="Bonn"] -> .bonn;
area[name="Frankfurt"] -> .frankfurt;
node.cafes(area.bonn) -> .cb;
node.cafes(area.frankfurt) -> .cf;
(.cf; .cb;) -> .cfb;
out .cfb;
```

Same query, but flattened:

```
area[name="Bonn"] -> .bonn;
area[name="Frankfurt"] -> .frankfurt;
( node[amenity="cafe"](area.bonn);
    node[amenity="cafe"](area.frankfurt); );
out;
```

1.4.6 And everything else

There are more available statements within Overpass QL, including:

- Recurse up/down relations — get items that are within relations in the current set and get relations the items in the input set are contained in
- "is in" — get areas the items in the input set are contained in
- Recurse along ways - get next, previous objects along way
- Fixed point — perform a block on an input set until it returns the same set

Since they have not been used in this reference implementation, they will not be discussed in detail. For extra information, see [wik19b].

2 Ingredients

2.1 Simply typed λ-calculus

For representing compositional terms, a version of the simply typed lambda calculus will be used.

2.1.1 Types

Let $\mathbb{T} \overset{\text{def}}{=} \{t', t'', t'''...\}$ be a countable set of symbols that we will call *type symbols*[4].

Definition 2.1. For a set of type symbols T, its *type closure* $\mathcal{T}(T) \subseteq (T \cup \{\to,), (\})^*$ is defined inductively:

- $\sigma \in T \implies \sigma \in \mathcal{T}(T)$
- $\sigma, \tau \in \mathcal{T}(T) \implies (\sigma \to \tau) \in \mathcal{T}(T)$

Convention 2.1. For clarity, braces shall be omitted in the notation for types by regarding the $\to$ operation as right-associative:

$$\sigma \to \tau \to \eta$$

will mean

$$(\sigma \to (\tau \to \eta))$$

Definition 2.2. The function $\mathsf{ts} : \mathcal{T}(\mathbb{T}) \to 2^{\mathbb{T}}$, which returns the set of type symbols used in a complex type, is defined inductively:

- $\sigma \in \mathbb{T} \implies \mathsf{ts}(\sigma) \overset{\text{def}}{=} \sigma$
- $\sigma = \sigma' \to \sigma'' \implies \mathsf{ts}(\sigma) \overset{\text{def}}{=} \mathsf{ts}(\sigma') \cup \mathsf{ts}(\sigma'')$

Definition 2.3. The types $\tau, \sigma \in \mathcal{T}(T)$ have *the same syntactic structure* when either:

- $\tau \in T \,\&\, \sigma \in T$
- $\tau = \tau' \to \tau'' \,\&\, \sigma = \sigma' \to \sigma''$ where τ' and σ', and τ'' and σ'' respectively have the same syntactic structure.

We will now introduce a notation for easily obtaining the types of items — $\|x\|$ shall mean "the type of x":

Definition 2.4. We call a set X *typed in* T when there is a function

$$\| \cdot \| : X \to \mathcal{T}(T)$$

that we call a *typing function*.

[4]See appendix A

Definition 2.5. If X, Y are sets of type symbols and $\varphi : X \to \mathcal{T}(Y)$, then $\hat{\varphi} : \mathcal{T}(X) \to \mathcal{T}(Y)$ is defined inductively:

$$\hat{\varphi}(\sigma) = \begin{cases} \varphi(\sigma), & \sigma \in X \\ \hat{\varphi}(\sigma') \to \hat{\varphi}(\sigma''), & \sigma = \sigma' \to \sigma'' \in \mathcal{T}(X) \end{cases}$$

2.1.2 Terms

To define λ terms with the types introduced above, a system similar to Church's $\lambda_{\to}$ [NG14, chap. 2.4] shall be used. It extends the original system with constants.

Let $\mathbb{V}^{\lambda} = \{\alpha', \alpha'', \alpha'''...\}$ be a countable set of symbols that we will call *variables* and $\mathsf{Const} = \{c', c'', c'''...\}$ be a countable set of symbols that we will call *constants*.

We can now define the set of *pre-typed λ-terms with constants C typed in T* (Λ_T^C).

Definition 2.6. Let $C \subset \mathsf{Const}$ be a set of constants, typed in T.

Λ_T^C is defined inductively:

- Constant:
$$c \in C \implies c \in \Lambda_T^C$$
- Variable:
$$v \in \mathbb{V}^{\lambda} \implies v \in \Lambda_T^C$$
- Application:
$$A, B \in \Lambda_T^C \implies (AB) \in \Lambda_T^C$$
- Abstraction:
$$v \in \mathbb{V}^{\lambda}, A \in \Lambda_T^C, \sigma \in \mathcal{T}(T) \implies (\lambda v : \sigma \Rightarrow A) \in \Lambda_T^C$$

Convention 2.2. In this text, chained abstractions will be squashed, i.e. $\lambda x_1 : T_1, x_2 : T_2, ..., y_n : T_n \Rightarrow M$ shall mean $\lambda x_1 : T_1 \Rightarrow (\lambda x_2 : T_2 \Rightarrow (...(\lambda x_n : T_n \Rightarrow M)...))$.

We will differentiate between *free variables* and *bound variables* in a term. Intuitively, bound variables are "bound" by abstraction, while free variables are not.

We will now formally define the function fv that gives us all free variables of a term.

Definition 2.7. Let $\mathsf{fv} : \Lambda_T^C \to 2^{\mathbb{V}^{\lambda}}$ be defined as:

$$\mathsf{fv}(M) = \begin{cases} \varnothing, & M = c \in C \\ \{v\}, & M = v \in \mathbb{V}^{\lambda} \\ \mathsf{fv}(A) \cup \mathsf{fv}(B), & M = (AB) \\ \mathsf{fv}(A) \backslash \{v\}, & M = (\lambda v : \sigma \Rightarrow A) \end{cases}$$

Definition 2.8. Statement, declaration, context, judgement [NG14, chap. 2.4]

- If $M \in \Lambda_T^C, \sigma \in \mathcal{T}(T)$, then $M : \sigma$ is called a *statement*. M is called a *subject* and σ is called a *type*.
- A statement with a variable as a subject is called a *declaration*.
- A set of declarations with different subjects is called a *context*.
- A *judgement* has the form $\Gamma \vdash M : \sigma$, where Γ is a context and $M : \sigma$ is a statement.

Furthermore, appending to contexts is defined as follows:

$$\Gamma \circ x : \sigma = \{y : \tau \in \Gamma \mid y \neq x\} \cup \{x : \sigma\}$$

The $\circ$ operation shall be regarded as left-associative throughout this text.
To define what it means for a term $M \in \Lambda_T^C$ to have a type, we will use derivation rules.

Definition 2.9. Derivation rules for typed λ-terms:

- Constant

$$\frac{c \in C}{\Gamma \vdash c : \|c\|}$$

- Variable

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

- Application

$$\frac{\Gamma \vdash A : \sigma' \to \sigma'' \qquad \Gamma \vdash B : \sigma'}{\Gamma \vdash (AB) : \sigma''}$$

- Abstraction

$$\frac{\Gamma \circ x : \tau \vdash A : \sigma}{\Gamma \vdash (\lambda x : \tau \Rightarrow A) : \tau \to \sigma}$$

Lemma 2.1. *Inversion lemma*:

- If $\Gamma \vdash c : \sigma$ for $c \in C$, then $\sigma = \|c\|$
- If $\Gamma \vdash x : \sigma$ for $x \in \mathbb{V}^\lambda$, then $x : \sigma \in \Gamma$
- If $\Gamma \vdash (AB) : \sigma''$, then for some σ':
  - $\Gamma \vdash A : \sigma' \to \sigma''$
  - $\Gamma \vdash B : \sigma'$
- If $\Gamma \vdash (\lambda x : \tau \Rightarrow A) : \nu$, then $\nu = \tau \to \sigma$ and $\Gamma \circ x : \tau \vdash A : \sigma$.

Proof. It follows immediately from the definition 2.9, since the cases are non-overlapping. □

Now, we will prove that every well-typed term has a unique type:

Proposition 2.1.

$$\Gamma \vdash M : \sigma' \;\&\; \Gamma \vdash M : \sigma'' \implies \sigma' = \sigma''.$$

Proof. We will assume that $\Gamma \vdash M : \sigma'$ and $\Gamma \vdash M : \sigma''$ and prove $\sigma' = \sigma''$ by induction over the construction of M. In every case, we will make use of lemma 2.1 to decompose the right-hand side of $\vdash$.

- $M = c \in C$:

  $\sigma' = \|c\| = \sigma''$

- $M = x$ for some $x : \sigma' \in \Gamma$ and $x : \sigma'' \in \Gamma$:

  Since Γ is a context (e.g. each subject appears only once), there is only one declaration with subject x, thus $\sigma' = \sigma''$.

- $M = (AB)$:

  We have $\Gamma \vdash (AB) : \sigma'$ and $\Gamma \vdash (AB) : \sigma''$. This means that

  $$\begin{aligned} \Gamma \vdash & A : \eta' \to \sigma' \\ \Gamma \vdash & B : \eta' \end{aligned}$$

  and

  $$\begin{aligned} \Gamma \vdash & A : \eta'' \to \sigma'' \\ \Gamma \vdash & B : \eta'' \end{aligned}$$

  But inductively A has a unique type, thus $\eta' \to \sigma' = \eta'' \to \sigma''$, therefore $\sigma' = \sigma''$.

- $M = (\lambda x : \tau \Rightarrow A)$, $\sigma' = \tau \to \eta'$ and $\sigma'' = \tau \to \eta''$.

  This means that $\Gamma \circ x : \tau \vdash A : \eta'$ and $\Gamma \circ x : \tau \vdash A : \eta''$. Inductively, $\eta' = \eta''$: therefore, $\sigma' = (\tau \to \eta') = (\tau \to \eta'') = \sigma''$.

□

Since we proved that types for terms are unique, we can now define a typing function (which will work on well-typed closed terms):

Definition 2.10. Now, $\|M\|$ for $M \in \Lambda_T^C$ is defined as follows:

$$\|M\| = \begin{cases} \sigma, & \text{if } \varnothing \vdash M : \sigma, \\ \neg!, & \text{otherwise.} \end{cases}$$

2.2 Pure CCG

The CCG formalism is used in a multitude of variants. The form presented here will be most basic and of little practical use: it will only allow composition rules and the only restriction will be a global upper bound on the arity.

The presentation of CCG in this section is based on the Vijay-Shanker CCG formalism [VW90].

This formalism is enough for understanding the parsing algorithms, and can be further extended to include features such as type-raising, arbitrary rule restrictions, slash modalities, category features, category variables, and feature variables in order to make it more practical.

Let $\text{Cat} = \{C', C'', C''', ...\}$ be a countable set of symbols that we will call *atomic categories*.

Definition 2.11. For a set of atomic categories τ, its *categorial closure* $\mathcal{C}(\tau)$ is defined as follows:

1. $A \in \tau \implies A \in \mathcal{C}(\tau)$
2. $X, Y \in \mathcal{C}(\tau) \implies (X/Y) \in \mathcal{C}(\tau)$
3. $X, Y \in \mathcal{C}(\tau) \implies (X\backslash Y) \in \mathcal{C}(\tau)$

Letters like A, B, C will be used to denote atomic categories, while letters like X, Y, Z will be used to denote complex categories (produced by rules 2 and 3).

Such expressions are called *categories*. The symbol $|$ will be used to denote any slash (i.e., when the distinction between $\backslash$ and $/$ does not matter). Categories will also be considered left-associative. Thus we can write $X_1|X_2|X_3|X_4$ to denote $(((X_1|X_2)|X_3)|X_4)$. It is useful to note that X_1 may be decomposed into its constituents until it becomes an atomic category: this means that every category can be written in the form $A|X_1|X_2|...|X_n$, where A is atomic. We say that n is the category's *arity*. A is called its *target* or return type, while $X_1...X_n$ are called its *arguments* or argument types.

Definition 2.12. Concatenating categories: If $X \in \mathcal{C}(\tau)$ and $Y = A|_1Y_1|_2Y_2...|_mY_m \in \mathcal{C}(\tau)$, then we define

$$X \circ_/ Y = X/A|_1Y_1|_2Y_2...|_mY_m$$
$$X \circ_\backslash Y = X\backslash A|_1Y_1|_2Y_2...|_mY_m$$

Definition 2.13. Breadth of category: for $X = A|X_1|X_2|...|X_n \in \mathcal{C}(\tau)$, $|X| = n$.

Definition 2.14. $G = \langle \Sigma, N, S, f, n \rangle$ is a *Combinatory Categorial Grammar*, where

- Σ is the (finite) set of *terminals*
- N is the (finite) set of *non-terminals* (atomic categories)
- $S \in N$ is the *target category*
- $f : \Sigma \to \hat{N}$, where $\hat{N}$ is the set of **finite** subsets of $\mathcal{C}(N)$, is the function for interpreting the terminals
- $n \in \mathbb{N}$ is the *maximum composition arity*

2.2.1 Derivations

We will now formalise the way in which CCGs generate strings of terminals. There are two approaches to do this: iterative derivation and derivation trees. Since they are both useful for different things, we will also show the equivalence between them.

We will now define the set of symbols that participate in the derivation process:

Definition 2.15. For a set of atomic categories τ, $S(\tau)$ is defined as the set of strings which contain:

- Complex categories enclosed in $[]$-brackets
- Terminals

It is defined formally as such:

$$S(\tau) \stackrel{\text{def}}{=} ([\mathcal{C}(\tau)] \cup \Sigma)^*$$

Let τ be a set of atomic categories. For iterative derivation, we will use strings in $S(\tau)$, i.e., strings of terminals intermixed with categories. To make our derivations equivalent to context-free grammar derivations, we use the square brackets ($[$, $]$) to delimit categories from the rest of the string.

Definition 2.16. For any string $\alpha \in S(\tau)$, where Σ is a finite alphabet of terminals, we define $\text{Cats}(\alpha)$ to be the set of all categories that can be found in α, namely:

- $\text{Cats}(\varepsilon) = \varnothing$
- for $a \in \Sigma$, $\text{Cats}(a\beta) = \text{Cats}(\beta)$
- for $X \in \text{Cats}(\tau)$, $\text{Cats}([X]\beta) = X \cup \text{Cats}(\beta)$

Definition 2.17. For a CCG G, we can construct a set $R \subset ([\mathcal{C}(N)]) \times S(N)$ of *rule instances*. Instead of $(\alpha, \beta) \in R$ we will write $\alpha \to_G \beta$.

- If $a \in \Sigma, X \in f(a)$, then
$$[X] \to_G a$$
- If $X/Y \in \mathcal{C}(N), Y|_1Z_1|_2Z_2 ... |_mZ_m \in \mathcal{C}(N), 0 \leqslant m \leqslant n$ then
$$[X|_1Z_1|_2Z_2...|_mZ_m] \to_G [X/Y][Y|_1Z_1|_2Z_2...|_mZ_m]$$
Furthermore, we call X/Y the *primary category* of the rule instance, while $Y|_1Z_1|_2Z_2...|_mZ_m$ is its *secondary category*. We also call X/Y the *left category* of the rule instance, and $Y|_1Z_1|_2Z_2...|_mZ_m$ its *right category*.
- If $X\backslash Y \in \mathcal{C}(N), Y|_1Z_1|_2Z_2...|_mZ_m \in \mathcal{C}(N), 0 \leqslant m \leqslant n$ then
$$[X|_1Z_1|_2Z_2...|_mZ_m] \to_G [Y|_1Z_1|_2Z_2...|_mZ_m][X\backslash Y]$$
Here, $X\backslash Y$ is the *primary* and *right* category, while $Y|_1Z_1|_2Z_2...|_mZ_m$ is the *secondary* and *left* category.

We will now define the notations $\Rightarrow_G$ for a derivation step and $\Rightarrow_G^*$ for multiple derivation steps.

Definition 2.18. Derivation process

- If $\beta \to \beta'$, then $\alpha\beta\gamma \Rightarrow_G \alpha\beta'\gamma$
- Let $\Rightarrow_G^*$ be the reflexive and transitive closure of $\Rightarrow_G$.
- If $\alpha \Rightarrow_G^* \beta$, we can write

  $$\mu : \alpha = \alpha_1 \Rightarrow_G \dots \Rightarrow_G \alpha_r = \beta$$

  and call μ a *derivation* for G. Then $\mathsf{Cats}(\mu) \overset{\text{def}}{=} \bigcup_{i=1}^{r} \mathsf{Cats}(\alpha_i)$ is the set of all categories used in μ.
- $L(G) := \{\alpha \in \Sigma^* \mid [S] \Rightarrow_G^* \alpha\}$

Convention 2.3. Whenever the grammar in use is clear from the context, we will write "$\to$, $\Rightarrow$, $\Rightarrow^*$" instead of respectively "$\to_G$, $\Rightarrow_G$, $\Rightarrow_G^*$.

Proposition 2.2. A CCG $G = \langle \Sigma, N, S, f, n \rangle$ is equivalent to an (albeit infinite) context-free grammar $G^C = \langle \Sigma, [\mathcal{C}(N)], R, [S] \rangle$.

Furthermore, let $\mu : \alpha_1 \Rightarrow \dots \Rightarrow \alpha_r$ be a derivation for G. We can construct a finite context-free grammar $G_\mu^C = \langle \Sigma, [\mathsf{Cats}(\mu)], R|_{[\mathsf{Cats}(\mu)]}, [S] \rangle$ that produces μ, and whose derivations are also derivations in G.

Proof. This follows because our definition of $\Rightarrow$ for G and the context-free definition of $\Rightarrow$ for G^C are the same. □

As per the given construction, there is no way for a CCG to generate the empty string.

Property 2.1. For any CCG G, $\varepsilon \notin L(G)$

Proposition 2.3. Concatenativity of CCG derivation:

For $\alpha \neq \varepsilon, \beta \neq \varepsilon$, the following two are equivalent:

- $\exists \alpha' \neq \varepsilon, \beta' \neq \varepsilon$ such that $\gamma = \alpha'\beta', \alpha \Rightarrow^* \alpha', \beta \Rightarrow^* \beta'$
- $\alpha\beta \Rightarrow^* \gamma$

Proof. We can use proposition 2.2:

- if $\alpha \Rightarrow^* \alpha'$ and $\beta \Rightarrow^* \beta'$, then $\alpha\beta \Rightarrow^* \alpha'\beta \Rightarrow^* \alpha'\beta' = \gamma$.
- if $\alpha\beta \Rightarrow^* \gamma, \alpha \neq \varepsilon, \beta \neq \varepsilon$, we can fix $\mu : \alpha\beta \Rightarrow \gamma_1 \Rightarrow \gamma_2 \dots \Rightarrow \gamma_t = \gamma$. Then μ is a valid derivation in the context-free grammar G_μ^C, which means α and β generate separate derivation subtrees, thus $\exists \alpha'\beta' = \gamma$ such that $\alpha \Rightarrow^* \alpha'$ and $\beta \Rightarrow^* \beta'$, which also holds for G.

□

Example 2.1. We will construct a simple grammar $G = \langle \Sigma, N, S, f, n \rangle$ that can generate the string `cities in Germany`.

$$
\begin{aligned}
\Sigma &\overset{\text{def}}{=} \{cities, in, Germany\} \\
N &\overset{\text{def}}{=} \{GSet\} \\
S &\overset{\text{def}}{=} GSet \\
f(x) &\overset{\text{def}}{=} \begin{cases} \{GSet\}, & \text{x = cities} \\ \{GSet\}, & \text{x = Germany} \\ \{GSet\backslash GSet/GSet\}, & \text{x = in} \end{cases} \\
n &\overset{\text{def}}{=} 1
\end{aligned}
$$

Here is a possible derivation process (categories to which we apply derivation rules on the next step are underlined):

$$
\begin{gathered}
\underline{[GSet]} \\
\Rightarrow \\
[GSet]\underline{[GSet\backslash GSet]} \\
\Rightarrow \\
\underline{[GSet]}[GSet\backslash GSet/GSet][GSet] \\
\Rightarrow \\
cities\ \underline{[GSet\backslash GSet/GSet]}[GSet] \\
\Rightarrow \\
cities\ in\ \underline{[GSet]} \\
\Rightarrow \\
cities\ in\ Germany
\end{gathered}
$$

Example 2.2. And now an even dumber grammar $G = \langle \Sigma, N, S, f, n \rangle$ that can generate strings like

`count to two, count to twice two, count to twice twice two` and so on.

$$
\begin{aligned}
\Sigma &\overset{\text{def}}{=} \{count, to, twice, two\} \\
N &\overset{\text{def}}{=} \{Action, Num\} \\
S &\overset{\text{def}}{=} Action \\
f(x) &\overset{\text{def}}{=} \begin{cases} \{Num\}, & \text{x = two,} \\ \{Action\}, & \text{x = count,} \\ \{Num/Num\}, & \text{x = twice,} \\ \{Action\backslash Action/Num\}, & \text{x = to.} \end{cases} \\
n &\overset{\text{def}}{=} 2
\end{aligned}
$$

Here is one possible derivation process:

$$
\begin{gathered}
\underline{[Action]} \\
\Rightarrow \\
[Action]\underline{[Action\backslash Action]} \\
\Rightarrow \\
[Action]\underline{[Action\backslash Action/Num]}[Num] \\
\Rightarrow \\
[Action][Action\backslash Action/Num]\underline{[Num/Num]}[Num] \\
\Rightarrow \\
[Action][Action\backslash Action/Num][Num/Num][Num/Num][Num] \\
\Rightarrow^* \\
count\ to\ twice\ twice\ two
\end{gathered}
$$

Derivation trees for CCG are ordered binary trees with categories labelling their internal nodes and terminals labelling their leaves.

We shall also consider the concept of *primary edges* in the context of derivation trees, in correspondence to the concept of *primary categories* in the context of derivations. Primary edges will be drawn with a thick line.

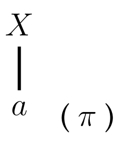
We will construct the set $\mathbb{T}_G$ of all derivation trees for a CCG G, as well as the functions $crown : \mathbb{T}_G \to S(N), root : \mathbb{T}_G \to \mathcal{C}(N)$.

If there exists $\tau \in \mathbb{T}_G$ such that $root(\tau) = X \in \mathcal{C}(N), crown(\tau) = \alpha$, we will denote $[X] \Rrightarrow \alpha$.

Derivation trees are constructed as follows:

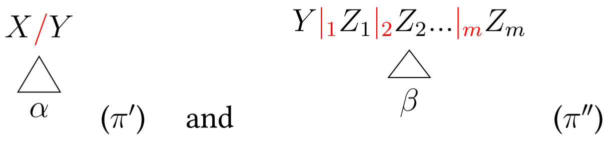
Definition 2.19. Derivation tree

- If $X \in \mathcal{C}(N)$ then $X \in \mathbb{T}_G, crown(X) = [X], root(X) = X$
- If $a \in \Sigma, X \in f(a)$, then

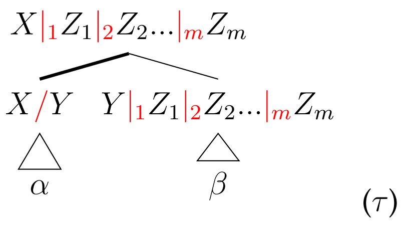


$\pi \in \mathbb{T}_G, crown(\pi) = a, root(\pi) = X$

- If

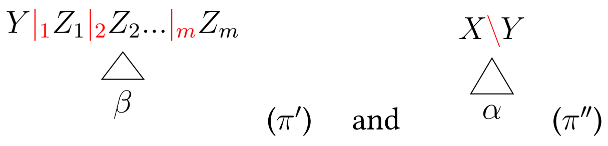


are derivation trees such that $root(\pi') = X/Y$, $root(\pi'') = Y|_1 Z_1|_2 Z_2 ...|_m Z_m$, $crown(\pi') = \alpha, crown(\pi'') = \beta$ where $m \leqslant n$, then

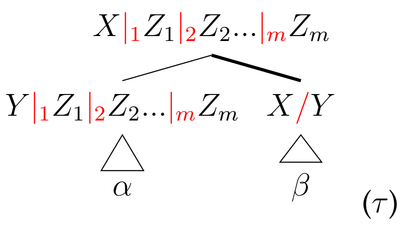


$\tau \in \mathbb{T}_G, root(\tau) = X|_1 Z_1|_2 Z_2 ...|_m Z_m, crown(\tau) = \alpha\beta.$

- If

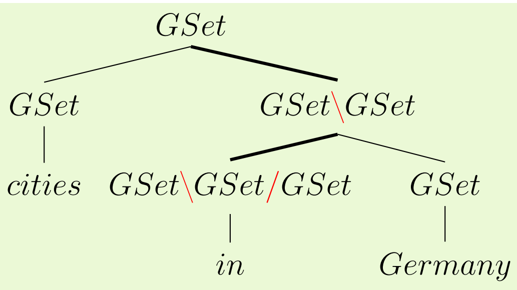


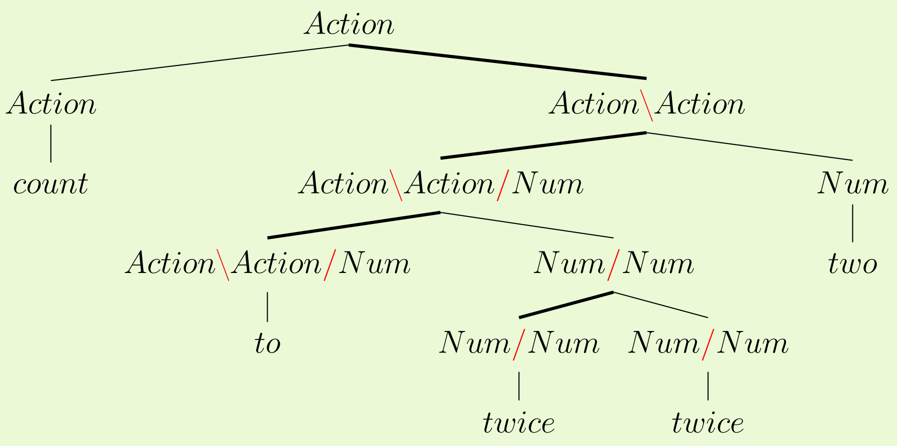
are derivation trees such that $root(\pi') = Y|_1 Z_1|_2 Z_2 ...|_m Z_m$, $root(\pi'') = X\backslash Y$, $crown(\pi') = \alpha, crown(\pi'') = \beta$ where $m \leqslant n$, then



$\tau \in \mathbb{T}_G, root(\tau) = X|_1 Z_1|_2 Z_2 ...|_m Z_m, crown(\tau) = \alpha\beta.$

Example 2.3. Here are the derivation trees for the derivations shown in example 2.1 and example 2.2 respectively:



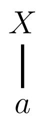

$Action$
$Action$
$Action\backslash Action$
$count$
$Action\backslash Action/Num$
Num
$Action\backslash Action/Num$
Num/Num
two
to
Num/Num
Num/Num
$twice$
$twice$

Proposition 2.4. Equivalence of derivation trees and derivations

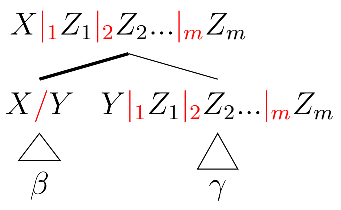
$[X] \Rightarrow^* \alpha \iff [X] \Rrightarrow \alpha$

Proof. First, let there exist a derivation tree for $[X] \Rrightarrow \alpha$. $[X] \Rightarrow^* \alpha$ can be proven by induction over constructing the tree.

- For the trivial cases, the derivation has 0 steps.
- For a tree

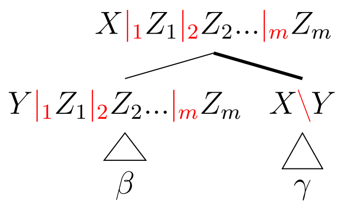


  and $X \in f(a)$, we have $\alpha = a$, $[X] \to a$.
- For a tree



  we have $\alpha = \beta\gamma$, and inductively $[Y|_1 Z_1|_2 Z_2 ...|_m Z_m] \Rightarrow^* \beta$ and $[X/Y] \Rightarrow^* \alpha$. Thus, $[X|_1 Z_1|_2 Z_2 ...|_m Z_m] \Rightarrow^* [X\backslash Y][Y|_1 Z_1|_2 Z_2 ...|_m Z_m] \Rightarrow^* \beta\gamma = \alpha$.

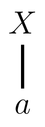
- For a tree

  we have $\alpha = \beta\gamma$, and inductively $[Y|_1Z_1|_2Z_2...|_mZ_m] \Rightarrow^* \beta$ and $[X\backslash Y] \Rightarrow^* \alpha$. Thus, $[X|_1Z_1|_2Z_2...|_mZ_m] \Rightarrow^* [Y|_1Z_1|_2Z_2...|_mZ_m][X\backslash Y] \Rightarrow^* \beta\gamma = \alpha$.

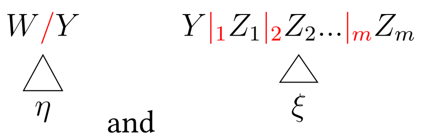
Second, we can prove the inverse implication (given a derivation, construct the derivation tree) by induction over derivation length. Let $\mu : [X] \Rightarrow \beta \Rightarrow^* \alpha$ be a derivation for G.

- $\beta = [X]$. Then this is the trivial reflexive case and X is a valid derivation tree in itself.
- $\beta = a \in \Sigma, X \in f(a)$. Then $\alpha = \beta$ and

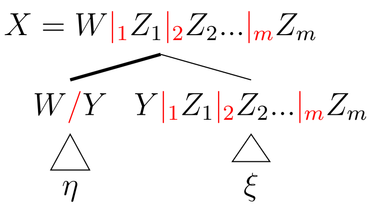
  

  is a derivation tree for $[X] \Rrightarrow a = \alpha$
- $X = W|_1Z_1|_2Z_2...|_mZ_m$ and $\beta = [W/Y][Y|_1Z_1|_2Z_2...|_mZ_m]$. According to proposition 2.3, there exist η, ξ such that:
  - $\alpha = \eta\xi$
  - $[W/Y] \Rightarrow^* \eta$
  - $[Y|_1Z_1|_2Z_2...|_mZ_m] \Rightarrow^* \xi$

  However, according to the inductive hypothesis, this means that we have the derivation trees

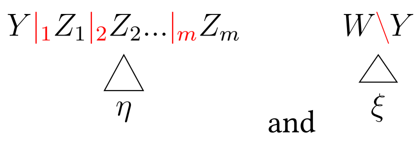
  Thus, by definition, we can construct the derivation tree

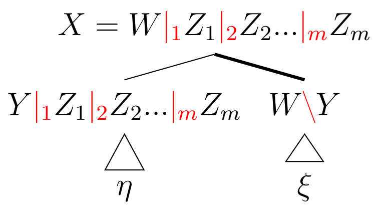
for $[X] \Rrightarrow \alpha$.

- $X = W|_1Z_1|_2Z_2...|_mZ_m$ and $\beta = [Y|_1Z_1|_2Z_2...|_mZ_m][W\backslash Y]$. According to proposition 2.3, there exist η, ξ such that:
  - $\alpha = \eta\xi$
  - $[Y|_1Z_1|_2Z_2...|_mZ_m] \Rightarrow^* \eta$
  - $[W\backslash Y] \Rightarrow^* \xi$

  However, according to the inductive hypothesis, this means that we have the derivation trees



  Thus, by definition, we can construct the derivation tree



  for $[X] \Rrightarrow \alpha$.

□

2.2.2 The Categorial CYK algorithm

This is the most straightforward algorithm for parsing a string with the help of CCG. It is easy to understand, but has exponential runtime with respect to the input word length.

It follows the same logic as the original CYK algorithm for context-free grammars, with the difference that it uses the asymmetric combinator rules from CCG to produce items.

Let $G = \langle \Sigma, N, S, f, n \rangle$ be a CCG and $w = w_1...w_k$ be a word.

The algorithm recursively builds a set P of *items* in the form (X, i, j), $X \in \mathcal{C}(N), 1 \leqslant i \leqslant j \leqslant k$ and aims to produce the item $(S, 1, k)$.

1. If $X \in f(w_i)$, then $(X, i, i) \in P$.
2. If $(X/Y, i, p) \in P$, $(Y|_1Z_1|_2Z_2...|_mZ_m, p+1, j) \in P$, then $(X|_1Z_1|_2Z_2...|_mZ_m, i, j) \in P$.
3. If $(X\backslash Y, p+1, j) \in P$, $(Y|_1Z_1|_2Z_2...|_mZ_m, i, p) \in P$, then $(X|_1Z_1|_2Z_2...|_mZ_m, i, j) \in P$.

To reason about the algorithm, we will use the following invariant:

$$[X] \Rightarrow^* w_i ... w_j \iff (X, i, j) \in P \tag{2.1}$$

Proposition 2.5. The algorithm is **correct**: if $(S, 1, k) \in P$, then $w \in L(G)$.

Proof. We have to prove the right-to-left direction of the invariant:

$$(X, i, j) \in P \implies [X] \Rightarrow^* w_i ... w_j \tag{2.2}$$

From which would follow $(S, 1, k) \in P \implies [S] \Rightarrow^* w_1 ... w_k$, which is what we need to prove.

So, let $(X, i, j) \in P$ and inductively suppose that (2.2) is true for any shorter substrings of w. There are 3 rules by which this item has appeared:

1. $i = j, X \in f(w_i)$: This means $[X] \to w_i \implies [X] \Rightarrow^* w_i = w_i ... w_j$
2. $i < j$ and
   - $X = W|_1 Z_1 |_2 Z_2 ... |_m Z_m$
   - $(W/Y, i, p) \in P$
   - $(Y|_1 Z_1 |_2 Z_2 ... |_m Z_m, p+1, j) \in P$

   for some $i \leqslant p < j, m \leqslant n$. By inductive hypothesis, $[W/Y] \Rightarrow^* w_i ... w_p$ and $[Y|_1 Z_1 |_2 Z_2 ... |_m Z_m] \Rightarrow^* w_{p+1} ... w_j$. Then, by the definition of $\to$ and by concatenativity of $\Rightarrow$, we have:

   $$[X] = [W|_1 Z_1 |_2 Z_2 ... |_m Z_m] \to [W/Y][Y|_1 Z_1 |_2 Z_2 ... |_m Z_m] \Rightarrow^* w_i ... w_p w_{p+1} ... w_j = w$$

3. $i < j$ and (analogous to 2)
   - $X = W|_1 Z_1 |_2 Z_2 ... |_m Z_m$
   - $(W \backslash Y, p+1, j) \in P$
   - $(Y|_1 Z_1 |_2 Z_2 ... |_m Z_m, i, p) \in P$

   for some $i \leqslant p < j, m \leqslant n$. By inductive hypothesis, $[W \backslash Y] \Rightarrow^* w_{p+1} ... w_j$ and $[Y|_1 Z_1 |_2 Z_2 ... |_m Z_m] \Rightarrow^* w_i ... w_p$. Then, by the definition of $\to$ and by concatenativity of $\Rightarrow$, we have:

   $$[X] = [W|_1 Z_1 |_2 Z_2 ... |_m Z_m] \to [Y|_1 Z_1 |_2 Z_2 ... |_m Z_m][W \backslash Y] \Rightarrow^* w_i ... w_p w_{p+1} ... w_j = w$$

□

Proposition 2.6. The algorithm is **complete**: if $w \in L(G)$, then $(S, 1, k) \in P$.

Proof. We have to prove the left-to-right direction of the invariant:

$$[X] \Rightarrow^* w_i...w_j \implies (X, i, j) \in P \tag{2.3}$$

From which would follow $[S] \Rightarrow^* w_1...w_k \implies (S, 1, k) \in P$, which is what we need to prove.

Let $\mu : [X] \Rightarrow \alpha \Rightarrow^* w_i...w_j$ be a derivation for G. We will prove (2.3) by induction over μ from right to left.

The only way for $[X]$ to appear to the left of a production is $[X] \rightarrow \alpha$ – we can now look at all ways for this to happen.

1. $\alpha = a \in \Sigma, X \in f(a)$. Then $w = \alpha, i = j, w_i = a$, which means $(X, i, i) = (X, i, j) \in P$.

2. $X = W|_1Z_1|_2Z_2...|_mZ_m$ and $\alpha = [W/Y][Y|_1Z_1|_2Z_2...|_mZ_m]$. According to proposition 2.3, there exists p such that

   - $[W/Y] \Rightarrow^* w_i...w_p$
   - $[Y|_1Z_1|_2Z_2...|_mZ_m] \Rightarrow^* w_{p+1}...w_j$

   However, according to the inductive hypothesis, this means that $(W/Y, i, p) \in P$ and $(Y|_1Z_1|_2Z_2...|_mZ_m, p + 1, j) \in P$. Thus, by the algorithm definition, $(X = W|_1Z_1|_2Z_2...|_mZ_m, i, j) \in P$.

3. $X = W|_1Z_1|_2Z_2...|_mZ_m$ and $\alpha = [Y|_1Z_1|_2Z_2...|_mZ_m][W\backslash Y]$ (analogous to 2.3). According to proposition 2.3, there exists p such that

   - $[Y|_1Z_1|_2Z_2...|_mZ_m] \Rightarrow^* w_i...w_p$
   - $[W\backslash Y] \Rightarrow^* w_{p+1}...w_j$

   However, according to the inductive hypothesis, this means that $(W\backslash Y, p + 1, j) \in P$ and $(Y|_1Z_1|_2Z_2...|_mZ_m, i, p) \in P$. Thus, by the algorithm definition, $(X = W|_1Z_1|_2Z_2...|_mZ_m, i, j) \in P$.

□

2.2.3 The Vijay-Shanker algorithm

This algorithm behaves like the categorial CYK algorithm for derivations that use only short categories. In order to pack longer categories in a way that would not result in exponential complexity, it uses some clever tricks.

A detailed explanation and proof can be found in the original paper [VW90].

Let $G = \langle\Sigma, N, S, f, n\rangle$ be a CCG and $w = w_1...w_k$ be a word.

Let $c' = max\{|X| \mid X \in f(a), a \in \Sigma\}$ and $c = max\{c', n\}$.

The algorithm uses two types of items:

1. $(X, i, j), X \in \mathcal{C}(N), 1 \leqslant i \leqslant j \leqslant k$
2. $(A, \xi, X, T, i, j, p, q), A \in N, T \in \{\backslash, /\}N, \xi \in \{\backslash, /\}, X \in \mathcal{C}(N), 1 \leqslant i \leqslant p \leqslant q \leqslant j \leqslant k$

As with the categorial CYK algorithm, we build a set P of items and aims to produce the item $(S, 1, k)$.

1. If $X \in f(w_i)$, then $(X, i, i) \in P$.
2. If $(X/B, i, t) \in P, (B|_1 Z_1|_2 Z_2 ... |_m Z_m, t+1, j) \in P$, $|X|_1 Z_1|_2 Z_2 ... |_m Z_m| \leqslant c$ then $(X|_1 Z_1|_2 Z_2 ... |_m Z_m, i, j) \in P$.
3. If $(A|_* X/B, i, t) \in P, (B|_1 Z_1|_2 Z_2 ... |_m Z_m, t+1, j) \in P$, $|A|_* X|_1 Z_1|_2 Z_2 ... |_m Z_m| > c$ then $(A, |_1, Z_1|_2 Z_2 ... |_m Z_m, /B, i, j, i, t) \in P$.
4. If $(A, \xi, X/B, T, i, t, p, q) \in P$, $(B|_1 Z_1|_2 Z_2 ... |_m Z_m, t+1, j) \in P$, $m > 1$, then $(A, \xi, Z_1|_2 Z_2 ... |_m Z_m, /B, i, j, i, t) \in P$.
5. If $(A, /, B, T, i, t, p, q) \in P$, $(B|_1 Z_1|_2 Z_2 ... |_m Z_m, t+1, j) \in P$, $m > 1$, then $(A, |_1, Z_1|_2 Z_2 ... |_m Z_m, T, i, j, p, q) \in P$.
6. If $(A, \xi, X/B, T, i, t, p, q) \in P$, $(B|_* Z, t+1, j) \in P$, then $(A, \xi, X|_* Z, T, i, j, p, q) \in P$.
7. If $(A, \xi, X/B, T, i, t, p, q) \in P$, $(B, t+1, j) \in P$, then $(A, \xi, X, T, i, j, p, q) \in P$.
8. If $(A, /, B, T, i, t, p, q) \in P$, $(B, t+1, j) \in P$, $(A|_1 X_1|_2 X_2 ... |_m X_m T, i, j) \in P$, then $(A|_1 X_1|_2 X_2 ... |_m X_m, i, j) \in P$.
9. If $(A, /, B, T, i, t, p, q) \in P$, $(B, t+1, j) \in P$, $(A, |_1 X_1|_2 X_2 ... |_m X_m T, T', i, j, r, s) \in P$, then $(A, |_1, X_1|_2 X_2 ... |_m X_m, T', i, j, r, s) \in P$.

The other 8 rules regarding backward composition ($2'$ through $9'$) are analogous to their forward counterparts, but with flipped indices.

The invariant here is composed of two parts:

- $(X, i, j) \in P \iff |X| \leqslant c \,\&\, [X] \Rightarrow^* w_i ... w_j$
- $(A, \xi, X, |_* B, i, j, p, q) \in P$ exactly when there exists $\nu \in \{\backslash, /\}, Y \in \mathcal{C}(N)$ such that
  - $1 \leqslant |Y| \leqslant c'$
  - $[A \nu Y \xi X] \Rightarrow^* w_i ... w_{p-1} [A \nu Y |_* B] w_{q+1} ... w_j$
  - $[A \nu Y |_* B] \Rightarrow^* w_p ... w_q$

For smaller grammars, which rarely generate very long categories during parsing, this algorithm is in fact slower than simple CYK-based alternatives. The strength of this algorithm comes when parsing automatically-generated grammars from treebanks.

For this reason, this algorithm will not be used. For a more in-depth overview, see [VW90].

2.3 Semantics within CCG

2.3.1 Extracting λ-terms from CCG derivations

A great strength of CCG comes from the ability to directly construct λ-terms from CCG derivations. This relies on the assumption that the target language has a compositional nature, namely that the semantics of any construction can be determined by composing the semantics of its constituents.

So, we shall to provide the semantics for the basic blocks (in our case the elements of Σ) in the form of λ-terms, and then define rules for composing those terms in accordance with a CCG derivation tree.

Definition 2.20. For a set of atomic categories N typed in Θ, we can extend the definition of the typing function $\|\cdot\|$ over their categorial closure $\|\cdot\| : \mathcal{C}(N) \to \mathcal{T}(\Theta)$ in the following way: $\|(X|Y)\| \overset{\text{def}}{=} \|Y\| \to \|X\|$.

This gives meaning to categories in the context of λ-calculus: $(X\backslash Y)$ refers to a function that returns a value of type X and takes an argument of type Y *on the right side*, while $(X\backslash Y)$ refers to a function that returns a value of type X and takes an argument of type Y *on the left side*.

Now that we have assigned types to categories, we can attach λ-terms to each terminal in addition to the categories we have attached via the function f. This will give us λ-terms in the leaves of each derivation tree, which can be composed naturally into λ-terms for all subtrees including the whole tree, giving us the semantic information for this parse.

To do this, we will use a function that maps each terminal/category pair to a λ-term. In order for the terms constructed on derivation trees to be correct and consistent, we will also want the terms of each node to be of the same type as its category.

Definition 2.21. Let Θ be a set of type symbols, $G = \langle \Sigma, N, S, f, n \rangle$ be a CCG, N be a set of atomic categories typed in Θ and C be a set of constants typed in Θ. A function

$$\psi : \{(a, X) \mid a \in \Sigma, X \in f(a)\} \to 2^{\Lambda_\Theta^C}$$

is called a *semantic function* for G.

While here we differentiate between *categories* and their *types*, it is often sufficient for types and categories to be the same set: when that is the case, the typing function is simply identity.

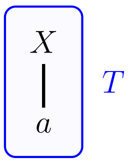
For simplicity, in most examples from here onward, the types of atomic categories will be the atomic categories themselves.

Now that we have our semantic function that can yield the possible terms for each *leaf* in a derivation tree, we will build the semantic function for the entire tree:

Definition 2.22. Let G be a CCG, ψ be a semantic function for G and T be a derivation tree for G.

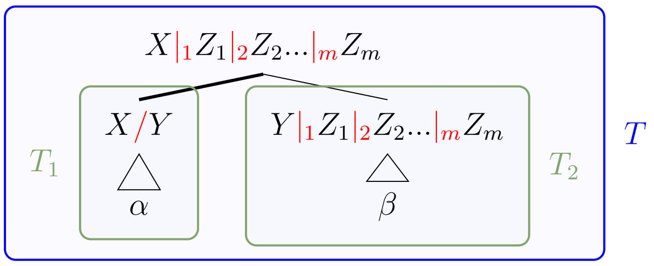
We will inductively define $sem_\psi(T) \subset \Lambda^C_\Theta$ with respect to T:

1. If T is



and $X \in f(a)$, then
$$sem_\psi(T) = \psi(a, X).$$

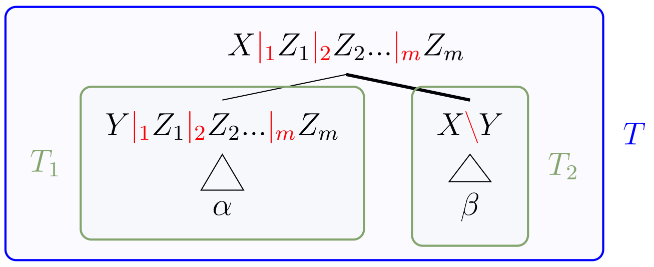
2. If T is



where $root(T_1) = X/Y$ and $root(T_2) = Y|_1 Z_1|_2 Z_2 ... |_m Z_m$ then
$$\begin{aligned} sem_\psi(T) = &\{\lambda z_m{:}\|Z_m\|, z_{m-1}{:}\|Z_{m-1}\|, ..., z_1{:}\|Z_1\| \Rightarrow M(N z_m z_{m-1} ... z_1) \\ &\mid M \in sem_\psi(T_1), N \in sem_\psi(T_2)\} \end{aligned}$$

for freshly-picked $z_1 ... z_m \in \mathbb{V}^\lambda \backslash \big(\mathsf{fv}(sem_\psi(T_1)) \cup \mathsf{fv}(sem_\psi(T_2))\big)$.

3. If T is



where $root(T_1) = Y|_1 Z_1|_2 Z_2 ... |_m Z_m$ and $root(T_2) = X\backslash Y$ then
$$\begin{aligned} sem_\psi(T) = &\{\lambda z_m{:}\|Z_m\|, z_{m-1}{:}\|Z_{m-1}\|, ..., z_1{:}\|Z_1\| \Rightarrow N(M z_m z_{m-1} ... z_1) \\ &\mid M \in sem_\psi(T_1), N \in sem_\psi(T_2)\} \end{aligned}$$

for freshly-picked $z_1 ... z_m \in \mathbb{V}^\lambda \backslash \big(\mathsf{fv}(sem_\psi(T_1)) \cup \mathsf{fv}(sem_\psi(T_2))\big)$.

Observe that since ψ returns valid λ-terms and everything returned by sem_ψ is constructed via basic λ-calculus operations, sem_ψ returns only valid λ-terms.

While it is easier to only compose closed terms, there are cases when free variables are useful. We will not restrict ourselves to closed terms and will therefore take some extra care to make this construction support free variables.

One way to do this is to have a global context for all terms that are given by the semantic function: this way we have a universal way for finding the type of each subterm, even if it contains free variables. Then, consistency between the terms returned by ψ and their categories can be defined cleanly:

Definition 2.23. A semantic function ψ is called *consistent in* a context Γ when

$$\forall a, X : M \in \psi(a, X) \implies \Gamma \vdash M : \|X\|.$$

Another observation is that the construction for sem_ψ includes no additional free variables other than the ones already present in the terms returned by ψ.

Proposition 2.7. If $G = \langle \Sigma, N, S, f, n \rangle$ is a CCG and ψ is a semantic function for G that is consistent in Γ, then for every valid derivation tree T:

$$M \in sem_\psi(T) \implies \Gamma \vdash M : \|root(T)\|$$

Proof. This can be proven by induction over the definition of sem_ψ.

Let $L \in sem_\psi(T)$ be a term produced by sem_ψ for an arbitrary derivation tree T. It has been constructed in one of the following ways:

- L was produced by case 1. Then $L \in \psi(a, X)$ for some $X \in f(a)$. ψ is consistent in Γ, so $\Gamma \vdash L : \|X\|$. Also, since $root(T) = X$, $\|root(T)\| = \|X\|$. Thus, $\Gamma \vdash L : \|root(T)\|$
- L was produced by case 2. Then, inductively we have:

$$\Gamma \vdash M : \|X/Y\| = \|Y\| \to \|X\| \tag{2.4}$$

and

$$\Gamma \vdash N : \|Y|_1 Z_1 |_2 Z_2 ... |_m Z_m\| = \|Z_m\| \to ... \to \|Z_1\| \to \|Y\| \tag{2.5}$$

Aside from that,

$$L = \lambda z_m{:}\|Z_m\|, z_{m-1}{:}\|Z_{m-1}\|, ..., z_1{:}\|Z_1\| \Rightarrow \underbrace{M(N z_m z_{m-1} ... z_1)}_{K} \tag{2.6}$$

Let $\Gamma' = \Gamma \circ z_m : \|Z_m\| \circ ... \circ z_2 : \|Z_2\| \circ z_1 : \|Z_1\|$. Since $z_1 ... z_m \notin \mathsf{fv}(M) \cup \mathsf{fv}(N)$ because we pick them as fresh variables, eq. (2.4) holds for Γ' in the same way it does for Γ.

So, $\Gamma' \vdash N z_m z_{m-1} ... z_1 : \|Y\|$ and then $\Gamma' \vdash \underbrace{M(N z_m z_{m-1} ... z_1)}_{K} : \|X\|$.

By the definition of $\vdash$ for the case of abstraction,

$$\Gamma \vdash \underbrace{(\lambda z_m{:}\|Z_m\|, z_{m-1}{:}\|Z_{m-1}\|, ..., z_1{:}\|Z_1\| \Rightarrow K)}_{L} : \|Z_m\| \to ... \to \|Z_1\| \to \|X\| \tag{2.7}$$

But $\|root(T)\| = \|X|_1 Z_1 |_2 Z_2 ... |_m Z_m\| = \|Z_m\| \to ... \to \|Z_1\| \to \|X\|$, therefore $\Gamma \vdash L : \|root(T)\|$ which is exactly what we sought to prove.

- L was produced by case 3. Since it is completely analogous to case 2, the proof will be omitted.

□

Given proposition 2.7, we know that all terms generated from CCG derivation trees will be consistent.

2.3.2 Composing phrases

The CCG formalism we have established allows a fixed set of terminals, which is usually chosen to be the set of all *tokens*, or words, in the target natural language. This works fairly well, because it is convenient to write rules at word level.

Sometimes, however, we come across fixed phrases, which come as several adjacent terminals but we want to regard as a single terminal.

Note: here we will use a modal forward slash ($/_\star$) which is defined in section 2.4.1. Its usage is not important for understanding the idea of this construction, so it may be regarded as a regular slash ($/$).

Example 2.4. Suppose we have a λ-term $parkinglot$ with type $Amenity$.

For example, we might want to assign the category $Amenity$ with term $parkinglot$ to the query `parking lot` that is composed of two terminals.

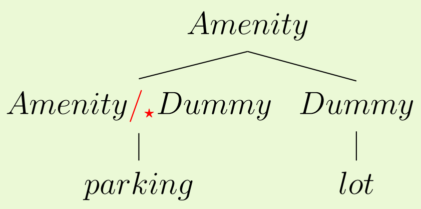
One way to do this is to set dummy categories to all but one of the terminals and have the main terminal accept the dummy categories as arguments:

- Add a new type/category $Dummy$.
- Choose a closed term M of type $Dummy$.
- Set

$$\begin{aligned} Amenity/_\star Dummy &\in f(parking) \\ Dummy &\in f(lot) \end{aligned}$$

$$\begin{aligned} \lambda d : Dummy \Rightarrow parkinglot &\in \psi(parking, Amenity/_\star Dummy) \\ M &\in \psi(lot, Dummy) \end{aligned}$$

This approach yields a derivation subtree for `parking lot` that has the correct category and semantics:



Convention 2.4. While defining a CCG $G = \langle \Sigma, N, S, f, n \rangle$ such that N is typed in T, the following assertions:

$$X \in f[\alpha_1 \alpha_2 ... \alpha_k]$$
$$M \in \psi[\alpha_1 \alpha_2 ... \alpha_k, X]$$

where $X \in \mathcal{C}(N)$ and M is a λ-term typed in T, have the following semantics:

- $\sigma_2, \sigma_3 ... \sigma_k \in T$, where σ_i is a new "dummy" type whose choice[5] depends on $\alpha_1, ... \alpha_k$ and i.
- $X_2, X_3, ..., X_k \in N$, where X_i is a new "dummy" category such that $\|X_i\| = \sigma_i$ and whose choice depends on σ_i.
- $M_2, M_3, ..., M_k$ are arbitrary "dummy" λ-terms whose types are respectively $\sigma_2, \sigma_3, ..., \sigma_k$
- New added categories to f:
  - $X /_\star X_2 /_\star X_3 /_\star ... /_\star X_k \in f(\alpha_1)$
  - For each $i \in \{2...k\}$, $X_i \in f(\alpha_i)$
- New added terms to ψ:
  - $(\lambda x_2 : \sigma_2, x_3 : \sigma_3, ..., x_k : \sigma_k \Rightarrow M) \in \psi(\alpha_1, X /_\star X_2 /_\star X_3 /_\star ... /_\star X_k)$
  - For each $i \in \{2...k\}$, $M_i \in \psi(\alpha_i, X_i)$

2.3.3 Shorthand for grammar specification

When manually defining grammars, it is tedious to write definitions for f and ψ using standard mathematical notation.

Thus, we will adopt a special syntax, assuming a CCG $G = \langle \Sigma, N, S, f, n \rangle$ with an arbitrarily large n.

[5]When using λ-terms that allow subtypes and a wildcard type, such as the type system used in Minipass, it is convenient to create new types by subtyping the identity type $* \rightarrow *$ and using $(\lambda x : * \Rightarrow x)$ as a dummy term.

The following assertion:

$$\begin{aligned} \alpha \ : \ & X_1 \ @ \ M_1 \\ & X_2 \ @ \ M_2 \\ & \quad \dots \\ & X_m \ @ \ M_m \end{aligned}$$

shall mean that:

$$\begin{aligned} \{X_1, X_2 ... X_m\} &\subseteq f[\alpha] \\ M_1 &\in \psi[\alpha, X_1] \\ M_2 &\in \psi[\alpha, X_2] \\ &\dots \\ M_m &\in \psi[\alpha, X_m] \end{aligned}$$

Here, the square brackets denote possible phrase matching (convention 2.4). It can be ignored or swept under the rug, which makes square brackets behave like round brackets.

Example 2.5. This meta-example shows a couple of rules to illustrate the syntax:

$$\begin{aligned} near \ &: \ GSet\backslash GSet/GSet \ @ \ \lambda from : GSet, things : GSet \Rightarrow \\ & \qquad\qquad and\ things\ (within\ nearDistance\ from) \\ german\ city \ &: \ GSet \ @ \ and\ inGermany\ city \\ & \quad GSet/Name \ @ \ \lambda n{:}Name \Rightarrow and\ (named\ n)\ (and\ inGermany\ city) \end{aligned}$$

The grammar it specifies is the following:

$$\begin{aligned} f[german\ city] &= \{GSet, GSet/Name\} \\ f(near) &= \{GSet\backslash GSet/GSet\} \end{aligned}$$

$$\begin{aligned} \psi(near, GSet\backslash GSet/GSet) &= \lambda from : GSet, things : GSet \Rightarrow \\ & \qquad and\ things\ (within\ nearDistance\ from) \\ \psi[german\ city, GSet] &= and\ inGermany\ city \\ \psi[german\ city, GSet/GSet] &= \lambda n{:}Name \Rightarrow and\ (named\ n)\ (and\ inGermany\ city) \end{aligned}$$

Here we assign two different categories with their respective semantics to the phrase `german city` – one that can be seen in the query

```
"fountains in a german city"
```

and another in the query

```
"the german city Frankfurt"
```

Also, most examples from here on will assume that grammars have a starting category $GSet$, to conform as close as possible to the implementation in section 3.2.

2.3.4 Ambiguity, spurious ambiguity and the Eisner normal form

One of the great problems of grammar systems such as CCG is ambiguity. By *ambiguity* we mean the phenomenon that a single sentence may be generated by many different derivation trees.

The simplest type of ambiguity is *lexical ambiguity*, where the given sentence actually has several meanings.

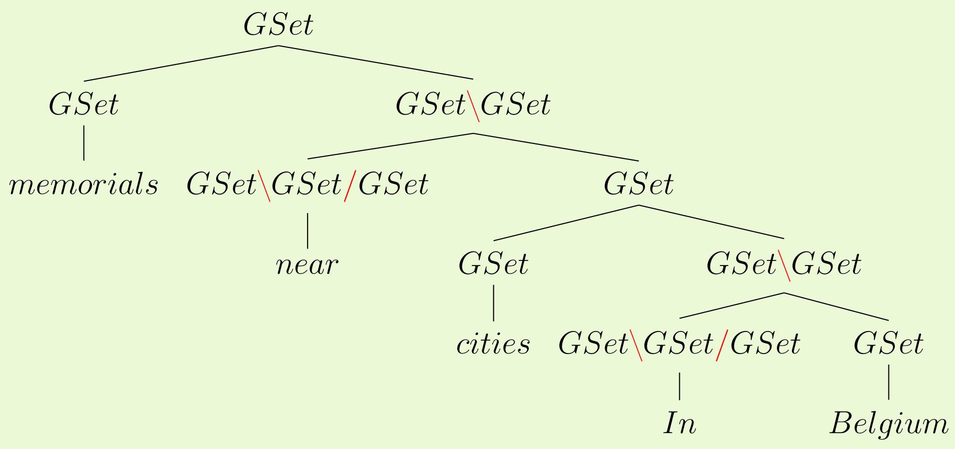
Example 2.6. For example, take the query

$$memorials\ near\ cities\ in\ Belgium$$

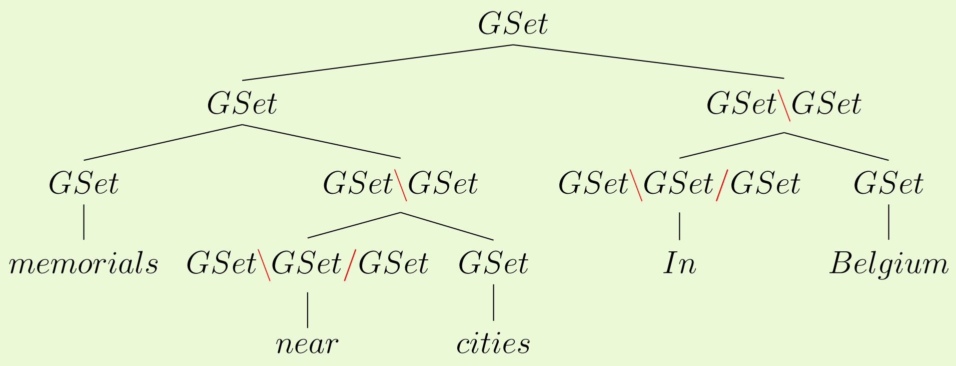
With the following simple grammar:

$$\begin{aligned} memorials &: GSet \\ cities &: GSet \\ near &: GSet\backslash GSet/GSet \\ in &: GSet\backslash GSet/GSet \end{aligned}$$

There are two possible derivation trees:



This parse means “memorials which are near cities, said *cities* being in Belgium”.



While this parse means “memorials which are near cities, said *memorials* being in Belgium”.

Both interpretations of the query are valid: a human would also interpret the query differently depending on their context and state of mind. Nevertheless, the set of objects returned for the two interpretations may be different (imagine a memorial just outside the border of Belgium that is near a city just inside the border).

The term *spurious ambiguity* within the context of CCG refers to generating different derivation trees *with the same meaning* for the same sentence. There is seldom any use in keeping such different trees in memory, so we would like to be able to leave only one tree for each distinct meaning.

The question is, what is “meaning”? Unlike Aristotle, we can choose an arbitrary definition to the word “meaning” by defining an equivalence relation between derivation trees — a predicate that tells us whether two trees mean the same thing.

Whenever using derivation trees to generate λ-terms, it would be most convenient to test whether they generate $\beta\eta$-equivalent terms. This might not guarantee the elimination of all semantically equivalent parses (two terms might be different but “do” the same thing depending on the interpretation) but will prune a significant part of them.

Since CCG derivations mimic function composition, one of the most common causes for spurious ambiguity arises from the associativity of composition: different trees may construct an equivalent λ-term by performing the same function compositions but in a different order.

Example 2.7. Let’s look at the query

$$small\ capital\ cities$$

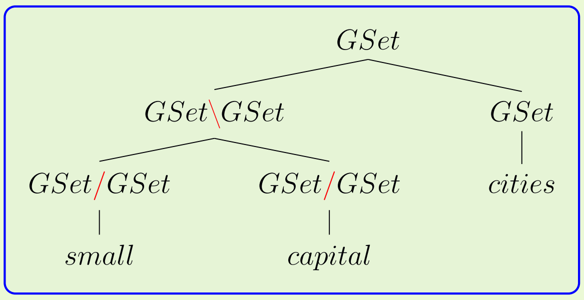
With the following simple grammar:

$$\begin{array}{rcl}
\mathit{small} & : & \mathit{GSet}/\mathit{GSet} \ @ \ \mathit{small} \\
\mathit{capital} & : & \mathit{GSet}/\mathit{GSet} \ @ \ \mathit{capital} \\
\mathit{cities} & : & \mathit{GSet} \qquad\quad\ @ \ \mathit{cities}
\end{array}$$

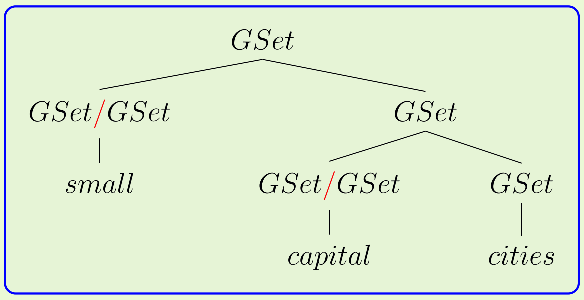
where $\|\mathit{small}\| = \mathit{GSet} \rightarrow \mathit{GSet}$, $\|\mathit{capital}\| = \mathit{GSet} \rightarrow \mathit{GSet}$ and $\|\mathit{cities}\| = \mathit{GSet}$.

Please take note that *small* (the token) is different from small (the λ-constant with type $\mathit{GSet} \rightarrow \mathit{GSet}$) and similarly for *capital* and *cities*.

There are two possible derivation trees:



T_1



T_2

Where by the definition of sem_ψ:

$$\begin{aligned}
(\lambda x{:}\mathit{GSet} \Rightarrow \mathit{capital}\ (\mathit{small}\ x))\ \mathit{cities} &\in \mathit{sem}_\psi(T_1) \\
\mathit{small}\ \mathit{capital}\ \mathit{cities} &\in \mathit{sem}_\psi(T_2)
\end{aligned}$$

However, $\mathit{sem}_\psi(T_1)$ can also be β-reduced to $\mathit{small}\ (\mathit{capital}\ \mathit{cities})$, so these two trees can be said to be a result of spurious ambiguity.

Most works that try to mitigate “spurious ambiguity” refer to this particular kind of spurious ambiguity — the kind arising from composition order.

A construction that completely eliminates compositional spurious ambiguity within pure CCG is Eisner Normal form [Eis96], which has been further refined by Hockenmaier et al. [HB10] to support most extensions shown in section 2.4 to various extents.

Since spurious ambiguity has not posed a significant problem for the grammars used in this thesis, this construction, while easy to implement, will not be discussed further.

2.4 ~~Hacks~~ Extensions of the CCG formalism

The CCG formalism described in section 2.2 is too weak to represent most phenomena in natural languages [Ste00]. Thus, several extensions have been developed to deal with them.

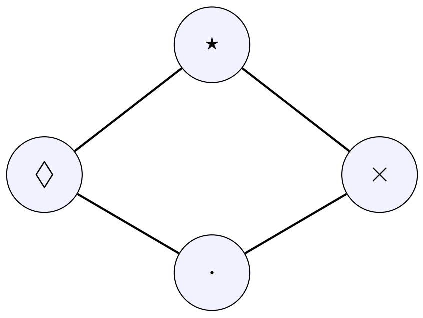
2.4.1 Rule restrictions or modalities

The only restriction on rules within the pure CCG formalism is the limit to the number of arguments (n). However, additional rule restrictions are often added in practice, which makes the grammar more expressive [KKS10].

The simplest method to implement this is to allow arbitrary rule restrictions, essentially creating a new formalism for each new target language. Another method is the so called *slash modalities*, which have been shown [BK03] to be equivalent to rule restrictions in terms of expressive power, and will be used here.

Definition 2.24. The elements of the set $\mathcal{M} = \{\star, \Diamond, \times, .\}$ are called *slash modalities*.

Slash modalities make up the following lattice:



For clarity, slash indices will be denoted with latin letters ($|_i$ denotes the i'th respective slash, including its direction and modality) while slash modalities will be denoted by Greek letters ($\backslash_\mu$ and $/_\mu$ denote the respective slash with modality μ).

Definition 2.25. We extend the definition of categorial closure to use modalities:

1. $A \in \tau \implies A \in \mathcal{C}(\tau)$
2. $X, Y \in \mathcal{C}(\tau), \mu \in \mathcal{M} \implies (X/_\mu Y) \in \mathcal{C}(\tau)$
3. $X, Y \in \mathcal{C}(\tau), \mu \in \mathcal{M} \implies (X\backslash_\mu Y) \in \mathcal{C}(\tau)$

Now, we restrict derivations according to modalities (in all cases, $m \leqslant n$):

- "Application"
$$[X] \to [X/_\mu Y][Y]$$
$$[X] \to [Y][X\backslash_\mu Y]$$
where $\mu \leqslant \star$ (effectively for any μ).

- "Harmonic composition"

$$[X/_{\eta}Z_1|_2Z_2...|_mZ_m] \rightarrow [X/_{\mu}Y][Y/_{\nu}Z_1|_2Z_2...|_mZ_m]$$

$$[X\backslash_{\eta}Z_1|_2Z_2...|_mZ_m] \rightarrow [Y\backslash_{\mu}Z_1|_2Z_2...|_mZ_m][X\backslash_{\nu}Y]$$

  where $\mu \leqslant \Diamond, \nu \leqslant \Diamond, \eta = \mu \wedge \nu$.

- "Crossing composition"

$$[X\backslash_{\eta}Z_1|_2Z_2...|_mZ_m] \rightarrow [X/_{\mu}Y][Y\backslash_{\nu}Z_1|_2Z_2...|_mZ_m]$$

$$[X/_{\eta}Z_1|_2Z_2...|_mZ_m] \rightarrow [Y/_{\mu}Z_1|_2Z_2...|_mZ_m][X\backslash_{\nu}Y]$$

  where $\mu \leqslant \times, \nu \leqslant \times, \eta = \mu \wedge \nu$.

In the above templates, modalities on the right-hand side stand for any modality that is less-than or equal (according to the modality lattice) to the one specified. The modalities on the left-hand side are produced by unifying (lowest common ancestor) the modalities on the right.

Example 2.8. For example, here are a few valid derivations:

$$[X] \rightarrow [X/_{\Diamond}Y][Y]$$

$$[X] \rightarrow [X/_{\times}Y][Y]$$

$$[X] \rightarrow [X/_{\cdot}Y][Y]$$

$$[X/_{\cdot}Z_1|_2Z_2...|_mZ_m] \rightarrow [X/_{\Diamond}Y][Y/_{\cdot}Z_1|_2Z_2...|_mZ_m]$$

$$[X/_{\cdot}Z_1|_2Z_2...|_mZ_m] \rightarrow [Y/_{\times}Z_1|_2Z_2...|_mZ_m][X\backslash_{\cdot}Y]$$

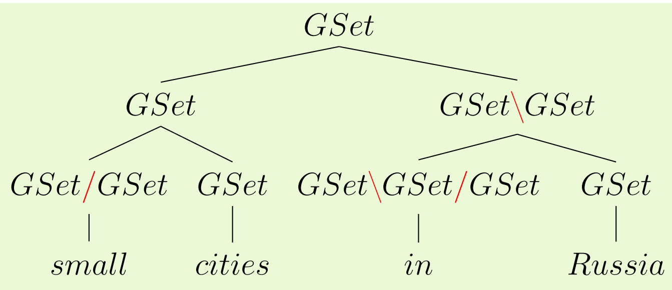
This way, one can restrict the ways in which a category would combine by choosing appropriate slash modalities.

When using slash modalities, the bare slashes ($\backslash$ and $/$) will actually mean $\backslash_{\cdot}$ and $/_{\cdot}$, being the most generic (they can combine with any other type of slash).

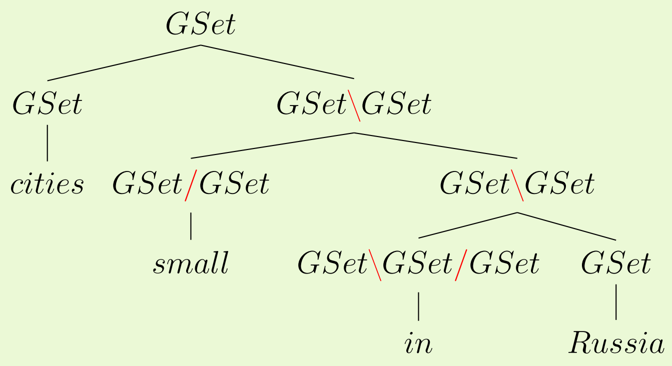
Example 2.9. Consider the following simple grammar:

$$\begin{aligned} small &: GSet/GSet \\ cities &: GSet \\ in &: GSet\backslash GSet/GSet \\ Russia &: GSet \end{aligned}$$

It generates the following query, which looks well:



However, it also generates this query, which is undesirable:



It appears that adjectives such as $small$ should not be allowed to combine in this way. This can be remedied by setting $small\ :\ GSet/_{\diamond}GSet$ in order to disallow crossing composition.

2.4.2 Categorial variables

This is a very powerful extension that adds simple unification based on variables to categorial derivation. To define it formally, let $\mathbb{V}^C = \{\kappa', \kappa'', \kappa''' \ldots\}$ be a countable set of symbols that we will call *categorial variables*.

Definition 2.26. We extend the definition of categorial closure to include variables:

1. $A \in \tau \implies A \in \mathcal{C}(\tau)$
2. $X, Y \in \mathcal{C}(\tau) \implies (X/Y) \in \mathcal{C}(\tau)$
3. $X, Y \in \mathcal{C}(\tau) \implies (X\backslash Y) \in \mathcal{C}(\tau)$
4. $\alpha \in \mathbb{V}^C \implies \alpha \in \mathcal{C}(\tau)$

For convenience, we will use the same notation as the function returning the free variables of λ-terms, for categories:

Definition 2.27. Let $\mathsf{fv} : \mathcal{C}(\tau) \to 2^{\mathbb{V}^C}$ be defined as:

$$\mathsf{fv}(Z) = \begin{cases} \varnothing, & Z \in \tau \\ \mathsf{fv}(X) \cup \mathsf{fv}(Y), & Z = (X|Y) \\ \{\alpha\}, & Z = \alpha \in \mathbb{V}^C \end{cases}$$

We now need to define *variable substitution* over categories in order to be able to easily manipulate variables.

Definition 2.28. Variable substitution over categories is defined as

$$Z[\alpha := W] = \begin{cases} Z, & Z \in \tau \\ (X[\alpha := W]|Y[\alpha := W]), & Z = (X|Y) \\ \beta, & Z = \beta \in \mathbb{V}^C, \beta \neq \alpha \\ W, & Z = \alpha \end{cases}$$

for $Z \in \mathcal{C}(\tau), W \in \mathcal{C}(\tau), \alpha \in \mathbb{V}^C$

Convention 2.5. If we have

$$X' = X[\alpha_1 := Y_1][\alpha_2 := Y_2]...[\alpha_n := Y_n],$$

then we will use

$$\xi = [\alpha_1 := Y_1][\alpha_2 := Y_2]...[\alpha_n := Y_n]$$

to denote that "ξ is the sequence of substitutions $[\alpha_1 := Y_1][\alpha_2 := Y_2]...[\alpha_n := Y_n]$".

The application of ξ on X will be denoted as

$$X' = \xi(X).$$

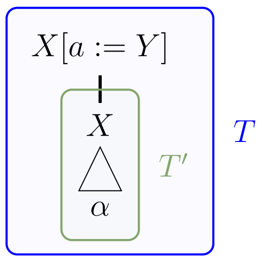
Having extended the notion of categorial closure to include categorial variables, we also add a family of derivation rules to deal with variables, namely:

$$\begin{tikzcd} X[\alpha := Y] \arrow[d, dash] \\ X \end{tikzcd}$$

for any $X \in \mathcal{C}(\tau), Y \in \mathcal{C}(\tau), \alpha \in \mathbb{V}^C$.

To define semantics for derivations with categorial variables, we need to extend our λ-calculus to include type-variables and type-variable substitution. For a more detailed explanation, see appendix B. Also, $\| \cdot \|$ for categories must be extended to injectively map categorial variables ($\mathbb{V}^C$) into type variables ($\mathbb{V}^T$) for variable substitution to work (and for category types that contain variables to match the types of their assigned terms).

Then, the semantics for the following derivation tree produced by variable substitution can be defined:



$$
\begin{aligned}
sem_\psi(T) &= \{M[\|a\| := Y] \mid M \in sem_\psi(T')\} \\
root(T) &= X[a := Y] \\
crown(T) &= crown(T') = \alpha
\end{aligned}
$$

This essentially allows us to generate any category and substitute it in place of variables at any time during derivation.

When parsing, we can observe that after processing all leaf nodes, there is no way to generate new variables while building the parse tree bottom-up. Thus, it is sufficient to eliminate variables by unification:

Definition 2.29. Two categories X and Y with categorial variables *unify* whenever we can find two sequences of substitutions $(a_1, X_1), (a_2, X_2), ..., (a_n, X_n)$ and $(b_1, Y_1), (b_2, Y_2), ..., (b_n, Y_n)$ such that

$$X[a_1 := X_1][a_2 := X_2][a_n := X_n] = Y[b_1 := Y_1][b_2 := Y_2][b_n := Y_n]$$

In order to implement parsing of CCGs with variables, we will modify the categorial CYK algorithm by replacing the rules (section 2.2.2) by the following:

1. If $X \in f(w_i)$, then $(X, i, i) \in P$.
2. If $(X/Y', i, p) \in P$, $(Y''|_1 Z_1|_2 Z_2 ...|_m Z_m, p+1, j) \in P$ and Y' and Y'' unify with substitutions $Y' \underbrace{[a_1 := Y'_1][a_2 := Y'_2][a_n := Y'_n]}_{\xi'} = Y'' \underbrace{[b_1 := Y''_1][b_2 := Y''_2][b_n := Y''_n]}_{\xi''}$, then $(\xi'(X)|_1 \xi''(Z_1)|_2 \xi''(Z_2)...|_m \xi''(Z_m)), i, j) \in P$.
3. If $(X\backslash Y', p+1, j) \in P$, $(Y''|_1 Z_1|_2 Z_2 ...|_m Z_m, i, p) \in P$ and Y' and Y'' unify with substitutions $Y' \underbrace{[a_1 := Y'_1][a_2 := Y'_2][a_n := Y'_n]}_{\xi'} = Y'' \underbrace{[b_1 := Y''_1][b_2 := Y''_2][b_n := Y''_n]}_{\xi''}$, then $(\xi'(X)|_1 \xi''(Z_1)|_2 \xi''(Z_2)...|_m \xi''(Z_m)), i, j) \in P$.

2.4.3 Coordination

Coordination refers to a technique that allows conjunctions (such as *and*, *or* etc.) to combine identical categories on both sides.

This can be emulated completely by using categorial variables and setting $and\ :\ \alpha\backslash\alpha/\alpha$[6] and similar.

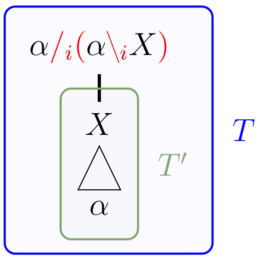
[6]In practice, one would need to set $and\ :\ X\backslash X/X, (X|\alpha)\backslash(X|\alpha)/(X|\alpha), ...$ in order to properly define semantics.

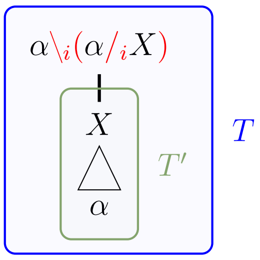
2.4.4 Type-raising

This extension adds a family of unary rules for constructing derivations [BB11, sec. 5.3.1], parametrised for any slash modality i:

$$\begin{tikzcd}[row sep=small] \alpha/_i(\alpha\backslash_i X) \arrow[d, dash] \\ X \end{tikzcd} \quad \text{(Forward type-raising)} \qquad \begin{tikzcd}[row sep=small] \alpha\backslash_i(\alpha/_i X) \arrow[d, dash] \\ X \end{tikzcd} \quad \text{(Backward type-raising)}$$

The semantics are defined as follows:





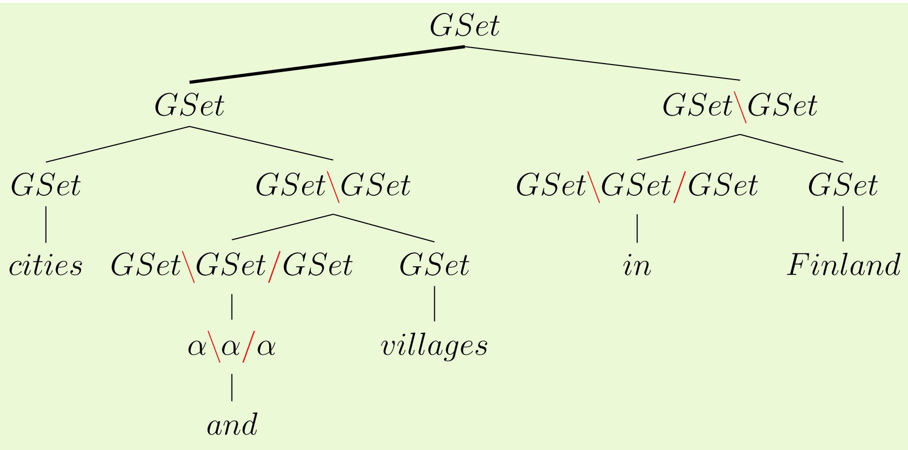
$$sem_\psi(T) = \{\lambda f{:}(X \to \alpha) \Rightarrow f M \mid M \in sem_\psi(T')\}$$

Since considering type-raising at any level would make parsing much slower, it is usually only considered on categories that appear in leaves.

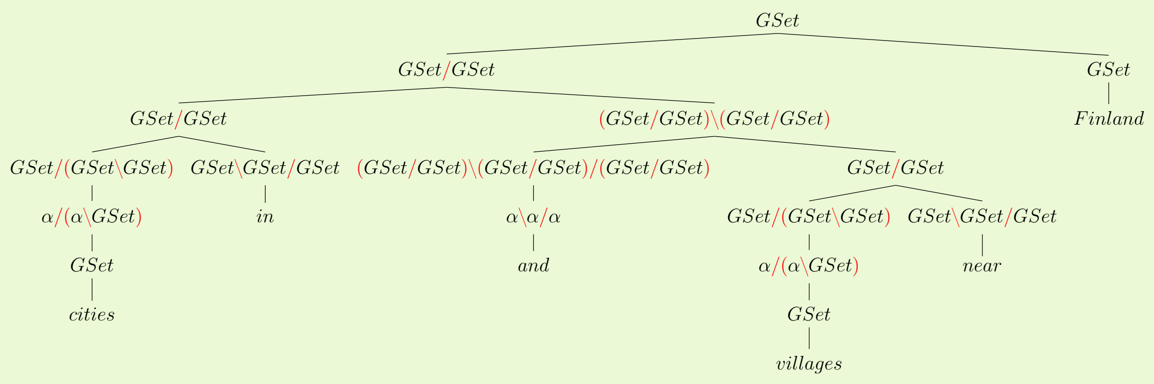
Example 2.10. Consider the following simple grammar:

$$\begin{aligned} cities &: GSet \\ villages &: GSet \\ Finland &: GSet \\ and &: \alpha\backslash\alpha/\alpha \\ in &: GSet\backslash GSet/GSet \\ near &: GSet\backslash GSet/GSet \end{aligned}$$

It permits derivations as such:



If we would like to be able to accept queries like "cities in and villages near Finland", we could permit type-raising:



2.4.5 Equification of categories

To reduce the size and of grammars and make them less complicated, it is convenient to regard not only identical categories as equal, but rather use an *equification relation* that can judge whether two categories must be regarded as equal.

One method is to explicitly state the equification rules in the grammar. It is also useful to make categories complex objects and infer the relation based on their structure, as will be done in the next two sections.

Definition 2.30. A relation $U \subseteq N \times N$ is called an *equification relation* for the set N if it is transitive and reflexive.

To make use of this relation for parsing, we must extend it over complex categories. One way to define such an extension is by uniformly propagating membership into U over complex categories:

Definition 2.31. If $U \subseteq N \times N$ is an equification relation for the set of atomic categories N, its extension over complex categories $\hat{U} \subseteq \mathcal{C}(N) \times \mathcal{C}(N)$ is defined inductively as follows:

- $X \in N \,\&\, Y \in N \,\&\, U(X, Y) \implies \hat{U}(X, Y)$

- $X = (W'|Z') \,\&\, Y = (W''|Z'') \,\&\, \hat{U}(W', W'') \,\&\, \hat{U}(Z', Z'') \implies \hat{U}(X, Y)$

Property 2.2. If $U \subseteq N \times N$ is an equification relation for the set of N, its extension $\hat{U}$ is an equification relation for $\mathcal{C}(U)$

Proof. The arguments for transitivity and reflexivity are both trivial. □

Another way to extend an equification relation is presented in section 2.4.7.

Now, we will modify the derivation rules from definition 2.17 to incorporate equification:

- If $a \in \Sigma, X \in f(a)$, then
$$[X] \to a$$
- If $X/Y' \in \mathcal{C}(N), Y''|_1 Z_1|_2 Z_2 ... |_m Z_m \in \mathcal{C}(N), 0 \leqslant m \leqslant n$ and $\hat{U}(Y', Y'')$, then
$$[X|_1 Z_1|_2 Z_2 ... |_m Z_m] \to [X/Y][Y|_1 Z_1|_2 Z_2 ... |_m Z_m]$$
- If $X \backslash Y' \in \mathcal{C}(N), Y''|_1 Z_1|_2 Z_2 ... |_m Z_m \in \mathcal{C}(N), 0 \leqslant m \leqslant n$ and $\hat{U}(Y', Y'')$, then
$$[X|_1 Z_1|_2 Z_2 ... |_m Z_m] \to [Y|_1 Z_1|_2 Z_2 ... |_m Z_m][X \backslash Y]$$

To implement parsing, we can replace the rules in the categorial CYK algorithm (section 2.2.2) by the following:

1. If $X \in f(w_i)$, then $(X, i, i) \in P$
2. If $(X/Y', i, p) \in P$, $(Y''|_1 Z_1|_2 Z_2 ... |_m Z_m, p+1, j) \in P$ and $\hat{U}(Y', Y'')$, then $(X|_1 Z_1|_2 Z_2 ... |_m Z_m, i, j) \in P$
3. If $(X \backslash Y', p+1, j) \in P$, $(Y''|_1 Z_1|_2 Z_2 ... |_m Z_m, i, p) \in P$ and $\hat{U}(Y', Y'')$, then $(X|_1 Z_1|_2 Z_2 ... |_m Z_m, i, j) \in P$

Note that the equification relation may be asymmetric and take advantage of the fact that its first argument is the tail of the primary category.

2.4.6 Equification by category features

Some models allow each atomic category to be extended with a set of *features* that refine its scope, and then define rules for unification of said features.

For example, we could assume a global set of features $Feat$ to be the set of finite functions $f : [a-zA-Z\_]^* \to [a-zA-Z\_]^*$ and instead of using $\mathcal{C}(\tau)$ directly, we use $\mathcal{C}(\tau \times Feat)$. For brevity, categories like $(V, \{(tense, past), (person, first)\})$ are written as $V[tense = past, person = first]$.

A common equification scheme is to check whether a union of the two feature functions, is also a function — this determines whether the categories actually equify.

This can be further extended by concepts such as *feature variables* (which allows the use of variables within feature expressions and performs variable substitution during unification) and embedding regular expressions.

2.4.7 Equification by subtyping

If the grammar will be used for generating typed λ-terms that use a subtyping system like the one described in section 2.5, the subtyping relation can be used as an equification relation.

If the desired semantics is to allow functions to only accept subtypes of their argument, some extra care needs to be taken, namely:

$$[X|_1 Z_1|_2 Z_2 \dots |_m Z_m] \rightarrow [X/Y_1][Y_2|_1 Z_1|_2 Z_2 \dots |_m Z_m]$$

$$[X|_1 Z_1|_2 Z_2 \dots |_m Z_m] \rightarrow [Y_2|_1 Z_1|_2 Z_2 \dots |_m Z_m][X\backslash Y_1]$$

if and only if

$$\|Y_2\| \leqslant \|Y_1\| \tag{2.8}$$

If we use an equification relation $U(X,Y) \overset{\text{def}}{\iff} \|Y\| \leqslant \|X\|$, it is not sufficient to use the uniform extension to complex categories (definition 2.31) because that would violate contravariance. Thus, we will use the following extension:

Definition 2.32. If $U \subseteq N \times N$ is an equification relation for the set of atomic categories N, its contravariant extension over complex categories $\overline{U} \subseteq \mathcal{C}(N) \times \mathcal{C}(N)$ is defined inductively as follows:

- $X \in N \;\&\; Y \in N \;\&\; U(X,Y) \implies \overline{U}(X,Y)$
- $X = (W'|Z') \;\&\; Y = (W''|Z'') \;\&\; \overline{U}(W'',W') \;\&\; \overline{U}(Z',Z'') \implies \overline{U}(X,Y)$

It can be easily seen that using $\overline{U}$ is equivalent to the condition in eq. (2.8) (given that $U(X,Y) \overset{\text{def}}{\iff} \|Y\| \leqslant \|X\|$).

The modification to the "derivation tree" construction and the semantic function is trivial. The proof for proposition 2.7 also holds (λ-terms with subtypes allow the argument to be a subtype of its required type).

Example 2.11. Imagine we want to be able to parse the following queries:

- `German cities less than 20km from Cologne`
- `German cities less than 20km north of Cologne`

For the sake of the example, subtrees for phrases like $German\ cities$ will be squashed and constants like $germanCity$, $within$ etc will be used without implementations[a].

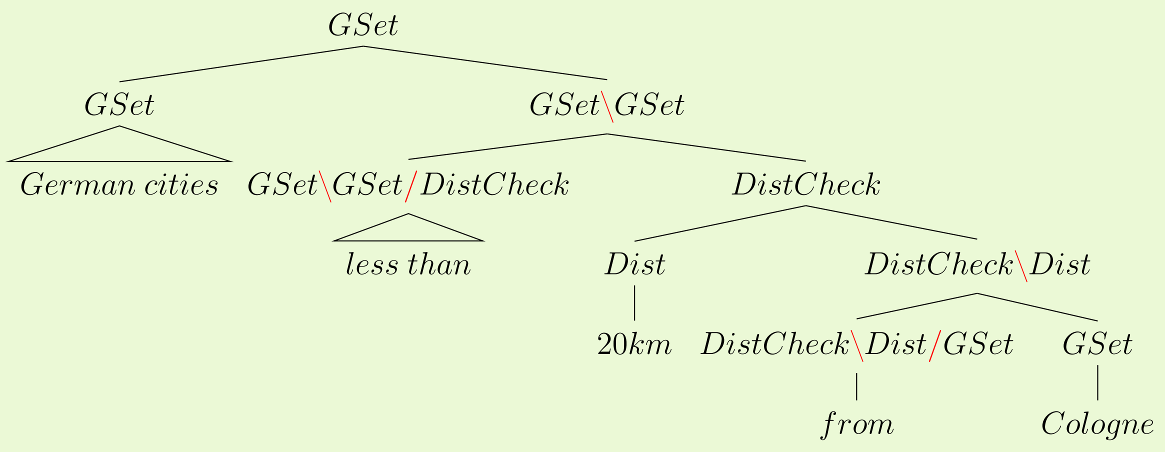
The grammar uses the following subtype definitions:

$$\begin{array}{rcl} DistCheck & <: & ((Dist \to GSet \to GSet) \to GSet) \\ NorthCheck & <: & ((Dist \to GSet \to GSet) \to GSet) \end{array}$$

and looks like this:

$$\begin{array}{rcllll}
German\ cities & : & GSet & @ & germanCity \\
Cologne & : & GSet & @ & cologne \\
20km & : & Dist & @ & twentykm \\
less\ than & : & GSet\backslash GSet/DistCheck & @ & \lambda b{:}DistCheck, s{:}GSet \Rightarrow \\
 & & & & \quad b\ (\lambda d{:}Dist, c{:}GSet \Rightarrow \\
 & & & & \quad\quad and\ s\ (within\ d\ c) \\
 & & & & \quad) \\
 & & GSet\backslash GSet/NorthCheck & @ & \lambda b{:}NorthCheck, s{:}GSet \Rightarrow \\
 & & & & \quad b\ (\lambda d{:}Dist, c{:}GSet \Rightarrow \\
 & & & & \quad\quad and\ s\ (withinNorth\ d\ c) \\
 & & & & \quad) \\
from & : & DistCheck\backslash Dist/GSet & @ & \lambda c{:}GSet, d{:}Dist \Rightarrow DistCheck[\\
 & & & & \quad \lambda f{:}(Dist \to GSet \to GSet) \Rightarrow f\ d\ c \\
 & & & &] \\
north\ of & : & NorthCheck\backslash Dist/GSet & @ & \lambda c{:}GSet, d{:}Dist \Rightarrow NorthCheck[\\
 & & & & \quad \lambda f{:}(Dist \to GSet \to GSet) \Rightarrow f\ d\ c \\
 & & & &]
\end{array}$$

Here is a derivation tree for one of the aforementioned queries:



The semantics of this tree is exactly the desired λ-term (after reduction):

$$and\ germanCity\ (within\ twentykm\ cologne)$$

And for the other query respectively:

$$and\ germanCity\ (withinNorth\ twentykm\ cologne)$$

Without subtyping, it would be difficult to discern the two categories $NorthCheck$ and $DistCheck$ without resorting to things like passing a dummy argument whose type encodes whether we are in one of the cases or in the other. Note that this cannot be done by simply using a type variable in the place of $DistCheck$ and $NorthCheck$ because of the different semantics (the categories $DistCheck$ and $NorthCheck$ combine with different contexts).

This example may seem strange to some (why would we want *less than* to know about $within$ and $withinNorth$? Perhaps it would be better to pair them with $from$?) but in reality the other way round is also strange — these semantics come from *less than* and $from$ *combined*. This violation of compositionality arises from the natural language itself, and subtyping is a useful "hack" to deal with it: we tag our type with additional information to denote that we are looking for something specific, and when we find it we can unwrap the type.

[a]This example uses minipass-like semantics for λ-terms to better illustrate the application. See section 3.2.

2.5 Simply typed λ-calculus with subtyping

In order to be able to represent the notion of generic and specific concepts, it is well-suited to use some sort of ordering relation over our types.

One such ordering relation is the *subtype* relation, as presented in [Pie02, chap. 15], and will be used here.

Example 2.12. The subtyping construction is particularly useful for domains like geographical queries — it lets us express relations such as the following:

$$\begin{gathered} Capital \leqslant City \leqslant GSet \\ Name \leqslant String \\ Distance \leqslant Number \end{gathered}$$

2.5.1 Type lattices

Definition 2.33. Meet and join:

We say that z is a *meet* of x and y, if:

$$z \leqslant x, z \leqslant y, \forall t \in X(t \leqslant x, t \leqslant y \implies t \leqslant z) \tag{2.9}$$

We say that z is a *join* of x and y, if:

$$x \leqslant z, y \leqslant z, \forall t \in X(x \leqslant t, y \leqslant t \implies z \leqslant t) \tag{2.10}$$

Definition 2.34. A *partial lattice* is a pair $\langle X, \leqslant \rangle$ where:

- $\leqslant$ is a partial-order relation
- for any $x, y \in X$, there is at most one meet and at most one join. They will be denoted as $x \wedge y$ (the meet of x and y) and $x \vee y$ (the join of x and y).

Definition 2.35. A partial lattice $\langle T, \leqslant \rangle$ where $T \subseteq \mathbb{T}$ is called a *type lattice.*

Whenever the relation is unambiguous, T and $\langle T, \leqslant \rangle$ will be used interchangeably: we could say "let T be a type lattice", which would refer to $\langle T, \leqslant \rangle$.

Definition 2.36. For a type lattice T, we extend[7] the relation $\leqslant$ over its type closure:

- $(\sigma \to \sigma') \leqslant (\tau \to \tau') \iff (\sigma' \leqslant \sigma)\&(\tau \leqslant \tau')$

Note that this definition assumes *contravariant arguments.* In essence, the semantics of $f \leqslant g$ is "wherever we can use g, we can also use f". Now, imagine that $f : \alpha' \to \beta'$, $g : \alpha'' \to \beta''$, and we have a function $h : (\alpha'' \to \beta'') \to \gamma$ that takes g as argument and produces something of type γ. Supposedly, h uses g somewhere in its body and supplies it with something of type α''.

Now, we supply f to h instead of g — this f needs to be able to take anything of type α'' and its subtypes as argument.

The only way to make this work is to make arguments contravariant, while making return values covariant.

Nevertheless, there exist systems (for example the Dart programming language [doc19]) where covariant arguments are allowed and the inconsistencies that arise from this decision are left to the user to look out for.

Lemma 2.2. If T is a type lattice and $\sigma = \sigma' \to \sigma'' \in \mathcal{T}(T), \tau = \tau' \to \tau'' \in \mathcal{T}(T)$, then:

1. η' is a join of σ' and τ' and η'' is a meet of σ'' and τ'' if and only if $\eta' \to \eta'' = \eta$ is a meet of σ and τ.
2. η' is a meet of σ' and τ' and η'' is a join of σ'' and τ'' if and only if $\eta' \to \eta'' = \eta$ is a join of σ and τ.

[7]To make this completely rigorous, $\mathcal{T}$ would have to be defined over *lattices*, not their carrier sets, so that the notion of "extending the relation" makes sense (because we actually create a new relation). These details, however, have been omitted for clarity.

Proof. We will prove both directions of (1) by applying equivalent transformations:

η being a meet of σ and τ is defined as:

$$\eta \leqslant \sigma, \eta \leqslant \tau, \forall \nu \in \mathcal{T}(T)(\nu \leqslant \sigma, \nu \leqslant \tau \implies \nu \leqslant \eta)$$

Now, expand the complex types:

$$(\eta' \to \eta'') \leqslant (\sigma' \to \sigma''),$$

$$(\eta' \to \eta'') \leqslant (\tau' \to \tau''),$$

$$\forall (\nu' \to \nu'') \in \mathcal{T}(T)((\nu' \to \nu'') \leqslant (\sigma' \to \sigma''), (\nu' \to \nu'') \leqslant (\tau' \to \tau'') \implies (\nu' \to \nu'') \leqslant (\eta' \to \eta''))$$

Then apply definition 2.36:

$$(\sigma' \leqslant \eta'), (\eta'' \leqslant \sigma'')$$

$$(\tau' \leqslant \eta'), (\eta'' \leqslant \tau'')$$

$$\forall (\nu' \to \nu'') \in \mathcal{T}(T)((\sigma' \leqslant \nu'), (\nu'' \leqslant \sigma''), (\tau' \leqslant \nu'), (\nu'' \leqslant \tau'') \implies (\eta' \leqslant \nu'), (\nu'' \leqslant \eta''))$$

Now, split the third condition (properties of universal quantification and implication) and rearrange the lines:

$$\sigma' \leqslant \eta', \tau' \leqslant \eta', \forall \nu \in \mathcal{T}(T)(\sigma' \leqslant \nu, \tau' \leqslant \nu \implies \eta' \leqslant \nu)$$

$$\eta'' \leqslant \sigma'', \eta'' \leqslant \tau'', \forall \nu \in \mathcal{T}(T)(\nu \leqslant \sigma'', \nu \leqslant \tau'' \implies \nu \leqslant \eta'')$$

This is exactly the condition that η' is a join of σ' and τ' and η'' is a meet of σ'' and τ''.

The argument for joins is dual to the one for meets and is, therefore, omitted for brevity. □

Proposition 2.8. If T is a type lattice, then $\mathcal{T}(T)$ is a partial lattice.

Proof. Let $\sigma, \tau \in \mathcal{T}(T)$. We will inductively (by their syntactic structure) prove that they have at most one join or meet.

First, let η_1 and η_2 be meets of σ and τ. Since $\leqslant$ is only defined if operands have the same syntactic structure and we have that $\eta_1 \leqslant \sigma, \eta_1 \leqslant \tau, \eta_2 \leqslant \sigma, \eta_2 \leqslant \tau$, they are all forced to have the same syntactic structure. Thus, there are two possibilities:

- $\sigma, \tau, \eta_1, \eta_2 \in T$:

  since T is a partial lattice, σ and τ have a unique meet. Thus, $\eta_1 = \eta_2$.

- $\sigma = \sigma' \to \sigma'', \tau = \tau' \to \tau'', \eta_1 = \eta_1' \to \eta_1', \eta_2 = \eta_2' \to \eta_2'$:

  Since η_1 and η_2 are meets of σ and τ, we have by lemma 2.2 that η_1' and η_2' are meets of σ' and τ', and that η_1'' and η_2'' are meets of σ'' and τ''.

  Inductively, $\eta_1' = \eta_2'$ and $\eta_1'' = \eta_2''$. Thus, $\eta_1 = \eta_1' \to \eta_1'' = \eta_2' \to \eta_2'' = \eta_2$.

The argument for joins is dual to the argument for meets and is again omitted. □

Property 2.3. For a type lattice T, and arbitrary $\sigma = \sigma' \to \sigma'' \in \mathcal{T}(T), \tau = \tau' \to \tau'' \in \mathcal{T}(T)$, we have:

$$\sigma \wedge \tau \simeq (\sigma' \vee \sigma'') \to (\tau' \wedge \tau'')$$
$$\sigma \vee \tau \simeq (\sigma' \wedge \sigma'') \to (\tau' \vee \tau'')$$

(where $\simeq$ means "either both sides are defined and equal or both are undefined")

Proof. This follows directly from lemma 2.2 and the proof of proposition 2.8. □

2.5.2 Terms

We will now extend the definition of λ-term given in section 2.1.1 to work with subtypes.

We need this in order to extend type systems with additional information, and to be able to dispose of the extra information when it is no longer needed.

So, for a of type lattice T, we will define the set of *pre-typed λ-terms with subtypes* ($\Lambda^C_{\leqslant T}$).

Definition 2.37. Let $C \subset \mathrm{Const}$ be a set of constants, typed in T.

$\Lambda^C_{\leqslant T}$ is defined inductively:

- Constant:
$$c \in C \implies c \in \Lambda^C_{\leqslant T}$$
- Variable:
$$v \in \mathbb{V}^\lambda \implies v \in \Lambda^C_{\leqslant T}$$
- Application:
$$A, B \in \Lambda^C_{\leqslant T} \implies (AB) \in \Lambda^C_{\leqslant T}$$
- Abstraction:
$$v \in \mathbb{V}^\lambda, A \in \Lambda^C_{\leqslant T}, \sigma \in \mathcal{T}(T) \implies (\lambda v : \sigma \Rightarrow A) \in \Lambda^C_{\leqslant T}$$
- Construction:
$$\sigma \in T, A \in \Lambda^C_{\leqslant T} \implies \sigma[A] \in \Lambda^C_{\leqslant T}$$

Definition 2.38. Derivation rules for λ-calculus with subtypes:

- Constant
$$\frac{c \in C}{\Gamma \vdash c : \|c\|}$$
- Variable
$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

- Application

$$\frac{\Gamma \vdash A : \sigma' \to \sigma'' \qquad \Gamma \vdash B : \tau \qquad \tau \leqslant \sigma'}{\Gamma \vdash (AB) : \sigma''}$$

- Abstraction

$$\frac{\Gamma \circ x : \tau \vdash A : \sigma}{\Gamma \vdash (\lambda x : \tau \Rightarrow A) : \tau \to \sigma}$$

- Construction via upcast

$$\frac{\Gamma \vdash A : \sigma \qquad \sigma \leqslant \tau}{\Gamma \vdash \tau[A] : \tau}$$

- Construction via downcast (dangerous, doesn't need to be included in most cases

$$\frac{\Gamma \vdash A : \sigma \qquad \tau \leqslant \sigma}{\Gamma \vdash \tau[A] : \tau}$$

Now we will show that every well-typed term has a unique type under a fixed context:

Proposition 2.9.

$$\Gamma \vdash M : \sigma' \,\&\, \Gamma \vdash M : \sigma'' \implies \sigma' = \sigma''$$

Proof. Assume $\Gamma \vdash M : \sigma'$ and $\Gamma \vdash M : \sigma''$ and prove this by induction over the construction of M:

- The proof for Constant, Variable, Application, and Abstraction is identical to the one for proposition 2.9
- $M = \sigma[A]$: this directly means that $\sigma' = \sigma = \sigma''$.

□

2.5.3 Type set subsumption

We will now formalise the notion of a type lattice being more general or more concrete than another type lattice.

Definition 2.39. If $\langle T, \leqslant' \rangle$ and $\langle S, \leqslant \rangle$ are type lattices, we say that $\langle T, \leqslant' \rangle \sqsupseteq \langle S, \leqslant \rangle$ if:

- $S \subseteq T$
- $\leqslant \subseteq \leqslant'$ and $\leqslant' \big|_{S \times S} = \leqslant$
- $\forall \tau \in (T \backslash S)((\exists \sigma \in S : \tau \leqslant \sigma) \veebar (\exists \sigma \in S : \sigma \leqslant \tau)$

(where $\veebar$ means logical "exclusive or")

Essentially, this says that T subsumes, or encompasses, S: any type in T is either a more concrete variant or a more general variant of some type in S.

If $\tau \in T$ is such that $(\exists \sigma \in S : \tau \leqslant \sigma)$, we will call τ *special*. Dually, if $\tau \in T$ is such that $(\exists \sigma \in S : \sigma \leqslant \tau)$, we will call τ *general*.

If we have $T \sqsupseteq S$, it would be convenient to be able to convert between them. The conversion from S to T is trivial. The conversion from T to S, however, involves discarding information.

Example 2.13. For example, let $S = \{Int, String\}$, supposing the obvious semantics behind those names. This is a nice type system — however, one might want to be able to encode extra information within types: this comes especially handy in the context of CCGs.

So, we add some extra types:

$$T = \{Int, String, Distance, Count, NamedEntity\}$$

We want $\leqslant$ to be defined as

$$Distance \leqslant Int, Count \leqslant Int, NamedEntity \leqslant String$$

Now while doing natural language parsing, we can use T. Afterwards we no longer need the extra information, so we could discard the extra types and return to S. To do this, we define a *squash function*.

Definition 2.40. $\omega : T \to S$ is a *squash function* for $\langle T, \leqslant' \rangle \sqsupseteq \langle S, \leqslant \rangle$, if:

- for each $\tau \in S$, $\omega(\tau) = \tau$
- it follows the subtyping relation:
$$\forall \sigma \in T(\omega(\sigma) \leqslant' \sigma \vee \sigma \leqslant' \omega(\sigma))$$
- it preserves the subtyping relation:
$$\forall \sigma, \tau \in T(\sigma \leqslant' \tau \implies \omega(\sigma) \leqslant \omega(\tau))$$

Property 2.4. Squash functions preserve the subtyping relation for complex types:

$$\forall \sigma, \tau \in \mathcal{T}(T)(\sigma \leqslant' \tau \implies \hat{\omega}(\sigma) \leqslant \hat{\omega}(\tau))$$

Proof. First, assume $\sigma \leqslant' \tau$.

The property is proven by induction over the construction of σ and τ.

- $\sigma \in T, \tau \in T$: this is true by definition.
- $\sigma = (\sigma' \to \sigma''), \tau = (\tau' \to \tau'')$: since $\sigma \leqslant' \tau$, we have that $\tau' \leqslant' \sigma'$ and $\sigma'' \leqslant' \tau''$. Therefore, inductively, $\hat{\omega}(\tau') \leqslant \hat{\omega}(\sigma')$ and $\hat{\omega}(\sigma'') \leqslant \hat{\omega}(\tau'')$. Thus,
$$\hat{\omega}(\sigma) = (\hat{\omega}(\sigma') \to \hat{\omega}(\sigma'')) \leqslant (\hat{\omega}(\tau') \to \hat{\omega}(\tau'')) = \hat{\omega}(\tau)$$

- in other cases, σ and τ have different syntactic structure and $\leqslant'$ is not defined. Since $\hat{\omega}$ preserves syntactic structure, $\hat{\omega}(\sigma)$ and $\hat{\omega}(\tau)$ will also have different syntactic structure, thus $\leqslant$ will be undefined for them as well.

$\square$

Property 2.5. If $\omega : T \to S$ is a squash function for $\langle T, \leqslant' \rangle \sqsupseteq \langle S, \leqslant \rangle$, then $\omega(\tau) = \min_{\leqslant}\{\sigma \in S \mid \tau \leqslant' \sigma\}$ for special elements $\tau \in T \backslash S$, and dually $\omega(\tau) = \max_{\leqslant}\{\sigma \in S \mid \sigma \leqslant' \tau\}$. for general elements $\tau \in T \backslash S$.

Proof. Without loss of generality[8], let τ be a special element in $T \backslash S$. Because of the "exclusive or" in definition 2.39, this means that $\exists \sigma \in S(\tau \leqslant' \sigma)$ and $\neg \exists \sigma \in S(\sigma \leqslant' \tau)$. Thus, for every $\sigma \in S$ that τ is comparable with, $\tau \leqslant' \sigma$.

Now, since $\omega(\tau)$ is comparable with τ, we have $\tau \leqslant' \omega(\tau)$.

Let $\sigma \in S$ be an arbitrary element such that $\tau \leqslant' \sigma$. By the preservation property of ω, we know that $\tau \leqslant' \omega(\tau) \leqslant \omega(\sigma) = \sigma$.

Thus, $\omega(\tau)$ is lower than any element that is comparable by $\leqslant$ with any element that is comparable by $\leqslant'$ with τ. This gives that $\omega(\tau)$ is the minimal element in S that is greater than τ. $\square$

Corollary 2.1. If there is a squash function $\omega : T \to S$ for $\langle T, \leqslant' \rangle \sqsupseteq \langle S, \leqslant \rangle$, it is unique in the sense that there is no other squash function for $\langle T, \leqslant' \rangle \sqsupseteq \langle S, \leqslant \rangle$.

Proof. The value of $\omega(\tau)$ is unambiguous in all cases:

- If $\tau \in S$, then $\omega(\tau) = \tau$.
- If $\tau \in T \backslash S$ and τ is special then $\omega(\tau) = \min_{\leqslant}\{\sigma \in S \mid \tau \leqslant' \sigma\}$.
- If $\tau \in T \backslash S$ and τ is general then $\omega(\tau) = \max_{\leqslant}\{\sigma \in S \mid \sigma \leqslant' \tau\}$.

$\square$

Property 2.6. The composition of two squash functions is a squash function, i.e. if:

- $\omega' : T' \to T$ is a squash function for $\langle T', \leqslant' \rangle \sqsupseteq \langle T, \leqslant \rangle$
- $\omega'' : T'' \to T'$ is a squash function for $\langle T'', \leqslant'' \rangle \sqsupseteq \langle T', \leqslant' \rangle$

Then $\omega \overset{\text{def}}{=} \omega' \circ \omega''$ is a squash function for $\langle T'', \leqslant'' \rangle \sqsupseteq \langle T, \leqslant \rangle$.

Proof. We will prove the squash function properties of ω one by one:

- for each $\tau \in T$, $\omega(\tau) = \tau$:

  We have that $\omega(\tau) = \omega'(\omega''(\tau))$. Since $T'' \sqsupseteq T' \sqsupseteq T$ implies $T'' \supseteq T' \supseteq T$, then $\tau \in T$ implies $\tau \in T'$. Also, $\omega''(\tau) = \tau$. Thus, $\omega'(\omega''(\tau)) = \tau$.

[8]The argument for general elements is dual to the one for special elements.

- it follows the subtyping relation:

  $$\forall \sigma \in T''(\omega(\sigma) \leqslant'' \sigma \vee \sigma \leqslant'' \omega(\sigma))$$

  From the same property for ω' and ω'', we have 2 cases:

  - Both relations are in the same direction, Without loss of generality $\omega''(\tau) \leqslant'' \tau$ and $\omega'(\omega''(\tau)) \leqslant' \omega''(\tau)$:
    This means that $\omega'(\omega''(\tau)) \leqslant'' \tau$.
  - The two relations are in different directions, Without loss of generality $\tau \leqslant'' \omega''(\tau)$ and $\omega'(\omega''(\tau)) \leqslant' \omega''(\tau)$:
    In this case, $\omega''(\tau)$ is both general for $T' \sqsupseteq T$ (because it is greatest in T' because of property 2.5) and special for $T' \sqsupseteq T$ (because it is less than $\omega'(\omega''(\tau)) \in T$).
    This means that $\tau \in T$, which makes the property trivial.

- it preserves the subtyping relation:

  $$\forall \sigma, \tau \in T''(\sigma \leqslant'' \tau \implies \omega(\sigma) \leqslant \omega(\tau))$$

  Let $\sigma, \tau \in T''$ be fixed such that $\sigma \leqslant'' \tau$. We have that $\omega''(\sigma) \leqslant' \omega''(\tau)$ because ω'' preserves $\leqslant''$. Subsequently, since ω' preserves $\leqslant'$, we have that $\omega'(\omega''(\sigma)) \leqslant \omega'(\omega''(\tau))$.

$\square$

We can now naturally extend squash functions to work over λ-terms and contexts:

Definition 2.41. If $\omega : T \to S$ is a squash function for $T \sqsupseteq S$ and C is typed in S, $\omega^\lambda : \Lambda^C_{\leqslant T} \to \Lambda^C_{\leqslant S}$ is defined inductively:

$$\omega^\lambda(A) = \begin{cases} c, & A = c \in C, \\ v, & A = v \in \mathbb{V}^\lambda, \\ \omega^\lambda(M)\omega^\lambda(N), & A = (MN), \\ \lambda x : \hat{\omega}(\sigma) \Rightarrow \omega^\lambda(M), & A = \lambda x : \sigma \Rightarrow M, \\ \hat{\omega}(\sigma)[M], & A = \sigma[M]. \end{cases}$$

Also, ω^γ, which operates on contexts is defined as follows:

$$\omega^\gamma(\Gamma) = \{x : \hat{\omega}(\sigma) \mid (x : \sigma) \in \Gamma\}$$

From here on ω may be used freely in place of ω^λ and ω^γ.

And now, the main proposition of this section, which tells us that when dropping a well-typed λ-term from an extended type system into a basic type system, it remains well-typed and has a useful type. In the implementation this is used for dropping from the user-defined type system used for CCG parsing into the Minipass type system (section 3.5.3).

Proposition 2.10. For a squash function $\omega : T \to S$, ω^λ is well-defined, namely

$$\Gamma \vdash A : \sigma \implies \omega^\gamma(\Gamma) \vdash \omega^\lambda(A) : \hat{\omega}(\sigma)$$

Proof. Let $\Gamma \vdash A : \sigma$. We will now prove the right-hand side by induction over the construction of A:

- $A = c \in C$. This means that $\|c\| = \sigma$ and thus $\sigma \in \mathcal{T}(S)$ because C is typed in S: therefore $\hat{\omega}(\sigma) = \sigma$. Also, we have that $\omega^\lambda(c) = c$, and so

  $$\Gamma \vdash c : \sigma \implies \omega^\gamma(\Gamma) \vdash c : \sigma \implies \omega^\gamma(\Gamma) \vdash \omega^\lambda(c) : \hat{\omega}(\sigma)$$

- $A = v \in \mathbb{V}^\lambda$. We have that $(v : \sigma) \in \Gamma$, thus $(v : \hat{\omega}(\sigma)) \in \omega^\gamma(\Gamma)$. But $\omega^\lambda(v) = v$, so $\omega^\gamma(\Gamma) \vdash \omega^\lambda(v) : \hat{\omega}(\sigma)$.

- $A = (MN)$, where $\Gamma \vdash M : (\eta' \to \sigma)$ and $\Gamma \vdash N : \eta$ where $\eta \leqslant \eta'$.

  Inductively, we know that

  $$\omega^\gamma(\Gamma) \vdash \omega^\lambda(M) : (\hat{\omega}(\eta') \to \hat{\omega}(\sigma)) \tag{2.11}$$

  $$\omega^\gamma(\Gamma) \vdash \omega^\lambda(N) : \hat{\omega}(\eta) \tag{2.12}$$

  And by property 2.4 we have that $\hat{\omega}(\eta) \leqslant \hat{\omega}(\eta')$.

  Finally, by the application rule of definition 2.38, we have that $\omega^\gamma(\Gamma) \vdash (\omega^\lambda(M)\omega^\lambda(N)) : \hat{\omega}(\sigma)$

- $A = (\lambda x : \tau \Rightarrow M)$, where $\Gamma' = \Gamma \circ (x : \tau) \vdash M : \eta$ and $\sigma = \tau \to \eta$.

  Inductively, we have that $\omega^\gamma(\Gamma') \vdash \omega^\lambda(M) : \hat{\omega}(\eta)$. However, $\omega^\gamma(\Gamma') = \omega^\gamma(\Gamma) \circ (x : \hat{\omega}(\tau))$, therefore

  $$\omega^\gamma(\Gamma) \vdash (\lambda x : \hat{\omega}(\eta) \Rightarrow \omega^\lambda(M)) : \hat{\omega}(\tau) \to \hat{\omega}(\eta) = \hat{\omega}(\sigma)$$

- $A = \sigma[M]$, where $\Gamma \vdash M : \tau$.

  Inductively, we have $\omega^\gamma(\Gamma) \vdash \omega^\lambda(M) : \hat{\omega}(\tau)$.

  Since either $\sigma \leqslant \tau$ (upcast) or $\tau \leqslant \sigma$ (downcast), by property 2.4 we have that either $\hat{\omega}(\sigma) \leqslant \hat{\omega}(\tau)$ or $\hat{\omega}(\tau) \leqslant \hat{\omega}(\sigma)$.

  Thus,

  $$\omega^\gamma(\Gamma) \vdash \hat{\omega}(\sigma)[\omega^\lambda(M)] : \hat{\omega}(\sigma).$$

$\square$

2.5.4 Subtype libraries

Definition 2.42. A statement in the form $\sigma <: \tau (\sigma, \tau \in \mathbb{T})$ is called a *subtype assertion.*

Since our definition of "subtype" doesn't allow us to subtype a complex type, we have to work around this by making σ be a complex type with the same syntactic structure as τ and then subtyping on the level of individual basic types it is made of. To do this, we need to be able to create a new term with the same syntactic structure of another term.

Example 2.14. Suppose we have the basic types Int and $String$, and we want to extend them.

For example, take the subtype assertion $F <: (Int \to String)$. Let $\tau = (Int \to String)$. We want F to correspond to a type σ such that $\sigma \leqslant \tau$. To do this, we clone the complex type τ, creating a new type with the same syntactic structure: $\sigma = (Int' \to String')$

Now we extend our subtyping relation to include $Int \leqslant Int', String' \leqslant String$, which indeed gives that $\sigma \leqslant \tau$.

Again, take note of the argument contravariance.

To remember that F stands for σ, we will iteratively build a function φ such that $\varphi(F) = \sigma$.

The following few definitions will formalise this process.

First, having a type τ, we need to be able to create a new type that is its clone (has the same syntactic structure), and is a subtype of τ.

To deal with contravariance, we will differentiate between a "subtype clone" ($\underline{clone}$) that clones a type into a new one that will be its subtype, and a "supertype clone", ($\overline{clone}$) that dually creates a "supertype" clone — in this way we will be able to use the $\overline{clone}$ in arguments and $\underline{clone}$ in return values.

To do this, we will use the mappings $\underline{\xi}$ and $\overline{\xi}$ that rename the basic types used in τ into new type symbols. To clone τ, we will recursively replace argument types and return value types with their respective clones. One way to deal with contravariance is the following:

Definition 2.43. Let $\langle T, \leqslant \rangle$ be a type lattice, $\tau \in \mathcal{T}(T)$ be a type and $\underline{\xi}, \overline{\xi}$ be mappings from a superset of $\mathrm{ts}(\tau)$ into $\mathbb{T}$ such that $range(\overline{\xi}) \cap range(\underline{\xi}) = \varnothing$.

$\overline{clone}_{\underline{\xi}\overline{\xi}}$ and $\underline{clone}_{\underline{\xi}\overline{\xi}}$ are defined by mutual recursion:

$$\underline{clone}_{\underline{\xi}\overline{\xi}}(\tau) = \begin{cases} \underline{\xi}(\tau), & \tau \in T \\ \overline{clone}_{\underline{\xi}\overline{\xi}}(\tau_1) \to \underline{clone}_{\underline{\xi}\overline{\xi}}(\tau_2), & \tau = \tau_1 \to \tau_2 \end{cases}$$

$$\overline{clone}_{\underline{\xi}\overline{\xi}}(\tau) = \begin{cases} \overline{\xi}(\tau), & \tau \in T \\ \underline{clone}_{\underline{\xi}\overline{\xi}}(\tau_1) \to \overline{clone}_{\underline{\xi}\overline{\xi}}(\tau_2), & \tau = \tau_1 \to \tau_2 \end{cases}$$

Property 2.7. If we have $\forall \eta (\underline{\xi}(\eta) \leqslant \eta)$ and $\forall \eta (\eta \leqslant \overline{\xi}(\eta))$, then for any τ:

- $\underline{clone}_{\underline{\xi}\overline{\xi}}(\tau) \leqslant \tau$

- $\tau \leqslant \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau)$

Proof. This property is proven by induction over the structure of τ:

- $\tau \in T$:
  - $\underline{clone}_{\underline{\xi}\bar{\xi}}(\tau) = \underline{\xi}(\tau) \leqslant \tau$
  - $\tau \leqslant \bar{\xi}(\tau) = \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau)$
- $\tau = \tau_1 \to \tau_2$:
  - Since inductively $\tau_1 \leqslant \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau_1)$ and $\underline{clone}_{\underline{\xi}\bar{\xi}}(\tau_2) \leqslant \tau_2$, by definition 2.36 we have that:
  $$\underline{clone}_{\underline{\xi}\bar{\xi}}(\tau) = \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau_1) \to \underline{clone}_{\underline{\xi}\bar{\xi}}(\tau_2) \leqslant \tau_1 \to \tau_2 = \tau.$$
  - Since inductively $\underline{clone}_{\underline{\xi}\bar{\xi}}(\tau_1) \leqslant \tau_1$ and $\tau_2 \leqslant \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau_2)$, by definition 2.36 we have that:
  $$\tau = \tau_1 \to \tau_2 \leqslant \underline{clone}_{\underline{\xi}\bar{\xi}}(\tau_1) \to \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau_2) = \overline{clone}_{\underline{\xi}\bar{\xi}}(\tau).$$

$\square$

Now we will define an iterative process for adding new types into a type lattice by subtyping existing types.

Suppose we have the subtype assertion $\sigma <: \tau$ with the intent of adding a new type "σ" to T such that $\sigma \leqslant \tau$. We are unable to add σ to T directly, since this would require subtyping a type with a different syntactic structure. So, instead we will construct another type $\varphi(\sigma) \in \mathcal{T}(T)$ whose constituent atomic types we add to T.

We will now define the operation $\oplus$ that does exactly that.

Afterwards, when we need to refer to τ, we will use $\varphi(\sigma)$.

Since we want to be able to define several new types one after the other, $\oplus$ will need to use the existing mapping φ to interpret existing manually-constructed types, and extend it with the new type.

Definition 2.44. Given

- a type lattice $\langle T, \leqslant\rangle$
- a type symbol $\sigma \in \mathbb{T}\backslash T$
- a mapping $\varphi : T \to \mathcal{T}(T)$
- a type τ such that $\hat{\varphi}(\tau)$ is defined

Let $\overline{\xi}$ and $\underline{\xi}$ be injections from $\mathsf{ts}(\hat{\varphi}(\tau))$ into $\mathbb{T}\backslash(T \cup \{\sigma\})$ such that $range(\overline{\xi}) \cap range(\underline{\xi}) = \varnothing$[9]. Now define the operation $\oplus$ as:

$$\langle\langle T, \leqslant\rangle, \varphi\rangle \oplus (\sigma <: \tau) \overset{\text{def}}{=} \langle\langle T', \leqslant'\rangle, \varphi'\rangle$$

where:

- $\rho = \underline{clone_{\underline{\xi}\overline{\xi}}}(\hat{\varphi}(\tau))$
- $T' = T \cup ((range(\underline{\xi}) \cup range(\overline{\xi})) \cap \mathsf{ts}(\rho))$
- $\leqslant'$ is the transitive and reflexive closure of $\leqslant \cup \{(\sigma', \tau') \in T' \times T \mid \underline{\xi}(\tau') = \sigma'\} \cup \{(\sigma', \tau') \in T \times T' \mid \overline{\xi}(\sigma') = \tau'\}$
- $\varphi' = \varphi \cup \{(\sigma, \rho)\}$

Note: in the cases where φ is irrelevant, it will be omitted as so:

$$\langle T, \leqslant\rangle \oplus (\sigma <: \tau) \overset{\text{def}}{=} \langle T', \leqslant'\rangle$$

Property 2.8. Objects constructed by $\oplus$ are correct type lattices.

Proof. We extend $\leqslant$ only by adding single edges (and their transitive and reflective closure) to new vertices, which creates a "tree-like" extension. There is no way to introduce a double join or meet, since this would require adding more than one edge to a new vertex. ☐

Lemma 2.3. If $\langle T, \leqslant\rangle$ is a type lattice, $\theta = (\sigma <: \tau)$ is a subtype assertion satisfying the conditions from definition 2.44 and $\langle T', \leqslant'\rangle = \langle T, \leqslant\rangle \oplus \theta$, then

$$T' \sqsupseteq T$$

Proof. Let $\underline{\xi}$ and $\overline{\xi}$ be the injections from definition 2.44. Now, we check the conditions for $T' \sqsupseteq T$:

- $T \subseteq T'$:

  this is true by definition ($T' = T \cup ((range(\underline{\xi}) \cup range(\overline{\xi})) \cap \mathsf{ts}(\rho))$) where $range(\underline{\xi})$ and $range(\overline{\xi})$ contain only fresh symbols (not in T).

- $\leqslant \subseteq \leqslant'$:

  $\leqslant'$ contains $\leqslant$ by definition.

- $\leqslant' \big|_{S \times S} = \leqslant$:

  we only add edges to $\leqslant$ that point to symbols that are not in T. Furthermore, the transitive and reflexive closures do not affect this condition because $\leqslant$ is already transitive and reflexive.

[9] $\overline{\xi}$ and $\underline{\xi}$ depend on τ, T and σ. If one wants to make their choice constrictive, they may, for example, assume that $\mathbb{T}$ is ordered, take the minimal elements that aren't in $T \cup \{\sigma\}$ and map to them in order.

- $\forall \tau \in (T' \backslash T)((\exists \sigma \in T : \tau \leqslant' \sigma) \veebar (\exists \sigma \in T : \sigma \leqslant' \tau)$:

  Fix an arbitrary $\tau \in (T' \backslash T)$.

  Since $T' \backslash T$ is comprised of elements from the ranges of $\underline{\xi}$ and $\overline{\xi}$, there are two options for τ:

  - $\tau \in range(\underline{\xi})$: this means that $\underline{\xi}(\sigma) = \tau$ for some $\sigma \in T$, but that also means that we have set $\tau \leqslant' \sigma$.
  - $\tau \in range(\overline{\xi})$: this means that $\overline{\xi}(\sigma) = \tau$ for some $\sigma \in T$, but that also means that we have set $\sigma \leqslant' \tau$.

  The exclusive "or" is satisfied because $range(\underline{\xi}) \cap range(\overline{\xi}) = \varnothing$.

□

Lemma 2.4. If $\langle T, \leqslant \rangle$ is a type lattice, $\theta = (\sigma <: \tau)$ is a subtype assertion and φ is a function, both of which satisfy the conditions from definition 2.44 and $\langle\langle T', \leqslant' \rangle, \varphi' \rangle = \langle\langle T, \leqslant \rangle, \varphi \rangle \oplus \theta$, then

$$\varphi'(\sigma) \leqslant' \hat{\varphi}(\tau).$$

Proof. Let $\underline{\xi}$ and $\overline{\xi}$ be the injections from definition 2.44.

We know that $\varphi'(\sigma) = \underline{clone}_{\underline{\xi}\overline{\xi}}(\hat{\varphi}(\tau))$.

Furthermore, $\forall \eta(\underline{\xi}(\eta) \leqslant' \eta)$ and $\forall \eta(\eta \leqslant' \overline{\xi}(\eta))$ by the construction of $\leqslant'$.

Then, by property 2.7 we have that $\varphi'(\sigma) \leqslant' \hat{\varphi}(\tau)$. □

Definition 2.45. A finite sequence of subtype assertions $\Theta_T = \theta_1 ... \theta_n$ is called a *subtype library* over T if the construction $\hat{\Theta}_T \overset{\text{def}}{=} \langle\langle T, \leqslant \rangle, \text{id}_T \rangle \oplus \theta_1 \oplus \theta_2 ... \oplus \theta_n$ is defined [10].

Also, we can define the function that gives us all the new type names asserted by the subtype library as $left(\Theta_T) = \{\sigma \mid \sigma <: \tau \in \Theta_T\}$

Proposition 2.11. If T is a type lattice and $\Theta_T = \theta_1 ... \theta_n$ is a subtype library over T, where $\hat{\Theta}_T = \langle\langle T', \leqslant' \rangle, \varphi \rangle$ the following properties hold:

- $dom(\varphi) = T \cup left(\Theta)$
- $\sigma \leqslant \tau \implies \sigma \leqslant' \tau$
- $\sigma <: \tau \in \Theta \implies \varphi(\sigma) \leqslant' \hat{\varphi}(\tau)$

Proof. We will check all conditions one by one:

- $dom(\varphi) = T \cup left(\Theta)$ (and φ is defined correctly):

  Initially, $dom(\varphi) = \text{id}_T$. For every $\sigma <: \tau \in \Theta$, we have added (σ, ρ) to φ for some ρ in $\mathcal{T}(T')$.

  Thus, we never violate the domain constraint for φ.

[10]When trying to redefine a previously defined type, the sequence is not a subtype library because of the restriction for σ in definition 2.44.

- $\sigma \leqslant \tau \implies \sigma \leqslant' \tau$:

  This is true because at each step we only add elements to the relation.

- $\sigma <: \tau \in \Theta \implies \varphi(\sigma) \leqslant' \hat{\varphi}(\tau)$:

  Let $\sigma <: \tau = \theta_i$.

  We have that $\langle\langle T', \leqslant'\rangle, \varphi\rangle = \langle\langle T, \leqslant\rangle, \mathrm{id}_T\rangle \oplus \theta_1 \oplus \dots \oplus \theta_i \oplus \dots \oplus \theta_n$

  Let $\langle\langle T^{(i)}, \leqslant^{(i)}\rangle, \varphi^{(i)}\rangle = \langle\langle T, \leqslant\rangle, \mathrm{id}_T\rangle \oplus \theta_1 \oplus \dots \oplus \theta_i \quad (T' = T^{(n)}, \leqslant' = \leqslant^{(n)})$.

  Since Θ is a valid subtype library, we know that $\hat{\varphi}^{(i-1)}(\tau) \in \mathcal{T}(T^{(i-1)})$.

  By lemma 2.4 $\varphi^{(i)}(\sigma) \leqslant^{(i)} \hat{\varphi}^{(i-1)}(\tau)$. Since φ is constructed incrementally, we also have $\varphi^{(n)}(\sigma) \leqslant^{(n)} \hat{\varphi}^{(n)}(\tau)$.

□

Lemma 2.5. If $\langle T, \leqslant\rangle$ is a type lattice, θ is a subtype assertion satisfying the conditions from definition 2.44 and $\langle T', \leqslant'\rangle = \langle T, \leqslant\rangle \oplus \theta$, then there is a squash function $\omega_\theta : T' \to T$.

Proof. Let $\underline{\xi}$ and $\overline{\xi}$ be the injections from definition 2.44.

Now, define the squash function ω_θ as:

$$\omega_\theta(\tau) \overset{\text{def}}{=} \begin{cases} \tau, & \tau \in T, \\ \sigma, & \tau = \underline{\xi}(\sigma), \\ \sigma, & \tau = \overline{\xi}(\sigma). \end{cases}$$

This is a correctly defined function because $\underline{\xi}$ and $\overline{\xi}$ are injections with non-overlapping ranges.

Now let us check the conditions for ω_θ being a squash function for $T' \sqsupseteq T$:

- For each $\tau \in T$, $\omega_\theta(\tau) = \tau$:

  This is true by the definition of ω_θ.

- It follows the subtyping relation:

  $$\forall \tau \in T(\omega_\theta(\tau) \leqslant' \tau \lor \tau \leqslant' \omega_\theta(\tau)).$$

  Let $\tau \in T'$. There are three possibilities for τ:

  - $\tau \in T$:

    $\omega_\theta(\tau) = \tau \leqslant' \tau$

  - $\tau = \underline{\xi}(\sigma)$ for some $\sigma \in T$:

    $\tau = \underline{\xi}(\sigma) \leqslant \sigma = \omega_\theta(\tau)$

  - $\tau = \overline{\xi}(\sigma)$ for some $\sigma \in T$:

    $\omega_\theta(\tau) = \sigma \leqslant \overline{\xi}(\sigma) = \tau$

- It preserves the subtyping relation:

$$\forall \sigma, \tau \in T'(\sigma \leqslant' \tau \implies \omega_\theta(\sigma) \leqslant \omega_\theta(\tau)).$$

  Let $\sigma \leqslant' \tau$ for some σ and τ. There are 4 possibilities:

  - $\sigma \in T, \tau \in T$:
    $\omega_\theta(\sigma) = \sigma \leqslant \tau = \omega_\theta(\tau)$.
  - $\sigma \in T, \tau \in (T' \backslash T)$:
    This means that $\omega_\theta(\sigma) = \sigma$ and $\omega_\theta(\tau) = \sigma$. Since $\leqslant$ is reflexive, $\sigma \leqslant \sigma$.
  - $\sigma \in (T' \backslash T), \tau \in T$:
    This means that $\omega_\theta(\tau) = \tau$ and $\omega_\theta(\sigma) = \tau$. Since $\leqslant$ is reflexive, $\tau \leqslant \tau$.
  - $\sigma \in (T' \backslash T), \tau \in (T' \backslash T)$:
    Let $\sigma' = \omega_\theta(\sigma)$ and $\tau' = \omega_\theta(\tau)$.

    Since an edge "$\sigma \leqslant' \tau$" wasn't explicitly added to $\leqslant'$, this means that they are in relation because of transitivity or reflexivity. The reflexive case is trivial.

    For each of σ and τ, we have added a single edge (plus transitive and reflective closure) in the relation. Therefore, having $\sigma \leqslant' \tau$, there is no σ'' such that $\sigma \neq \sigma'' \leqslant' \sigma$ and no τ'' such that $\tau \leqslant' \tau'' \neq \tau$.

    Since σ' is the least element in T such that $\sigma \leqslant' \sigma'$ and τ' is the greatest element in T such that $\tau' \leqslant' \tau$ (see property 2.5), we must have $\sigma \leqslant' \sigma' \leqslant \tau' \leqslant' \tau$.

□

The following proposition tells us that we can drop from an extended type system into a basic type system and constructs a mapping that does this in a sensible way (for each general type in the extended type system we take the most general possible basic type, and for each special type we take the most special possible basic type).

Proposition 2.12. If $\langle T, \leqslant \rangle$ is a type lattice and $\Theta_T = \theta_1 ... \theta_n$ is a subtype library over T, where $\hat{\Theta}_T = \langle \langle T', \leqslant' \rangle, \varphi \rangle$, then $T' \sqsupseteq T$. Furthermore, there is a squash function $\omega_\Theta : T' \to T$.

Proof. Let

$$\langle T_i, \leqslant_i \rangle = \langle T, \leqslant \rangle \oplus \theta_1 \oplus \theta_2 \oplus ... \oplus \theta_i$$

This property shall be proven by induction over the iterative process that constructs $T' = T_n$, while also building a squash function ω_i for each iteration:

- for T_0, we trivially know that $T \sqsupseteq T$ with an identity squash function $\omega_0(\sigma) = \sigma$.
- for T_{i+1}, we will define $\omega_{i+1}(\sigma) = \omega_i(\omega_{\theta_{i+1}}(\sigma))$.
  - Inductively we have that $T_i \sqsupseteq T$ and a squash function $\omega_i : T_i \to T$.
  - By lemma 2.3 we know that $T_{i+1} \sqsupseteq T_i$.

- By the transitivity[11] of $\sqsupseteq$, we have that $T_{i+1} \sqsupseteq T$.
- By lemma 2.5, $\omega_{\theta_{i+1}} : T_{i+1} \to T_i$ is a well-defined squash function.
- By property 2.6, ω_{i+1} is a squash function for $T_{i+1} \sqsupseteq T$.

□

The squash function we constructed here is unique by corollary 2.1.

2.5.5 Type-tagged terms and type propagation

For the sake of performing optimisations, it is convenient to store extra information within types. For example, we can extend our Num type to include a tag that says whether the value this type is attached to is a constant. This information can then be used to prune entire subterms whose value is constant.

In order to manipulate subterm types easily, we will introduce a construction called *type-tagged λ-terms*, which essentially means attaching a type to every subterm.

We will then define the functions tag and untag that allow us to convert typed λ-terms into type-tagged λ-terms and vice versa, and the function propagate that propagates type information throughout the term.

Example 2.15. For simplicity, regard the world of arithmetic expressions with a single type Num, numeric constants and the term $plus$ of type $Num \to Num \to Num$.

We will implement a very simple optimisation: constant folding.

Take the following term:

$$plus\ x\ (plus\ 3\ 5)$$

We shall regard the type of every subterm:

$$\begin{aligned}&((plus^{Num\to Num\to Num}\ x^{Num})^{Num\to Num}\\ &\quad((plus^{Num\to Num\to Num}\ 3^{Num})^{Num\to Num}\ 5^{Num})^{Num})^{Num}\end{aligned}$$

Now, we will extend the type system to the following lattice:

$$\begin{tikzcd}
 & & & Num \arrow[dlll, no head] \arrow[dll, no head] \arrow[dl, no head] \arrow[dr, no head] & & \\
C0 & C1 & C2 & \dots & C69 & \dots
\end{tikzcd}$$

Since numeric constants like 5 still bear the Num type, we will attach a *type-tag* to each

[11] Which can be trivially proven

subterm in order to be able to give them more concrete types:

$$\langle\langle\langle plus, Num \to Num \to Num\rangle\,\langle x, Num\rangle, Num \to Num\rangle$$
$$\langle\langle\langle plus, Num \to Num \to Num\rangle\,\langle 3, Num\rangle, Num \to Num\rangle\,\langle 5, Num\rangle, Num\rangle, Num\rangle$$

We want to apply the following "rewrite rules" in the form of a mapping ρ:

$$\rho(M) \overset{\text{def}}{=} \begin{cases} \langle i, C_i\rangle, & M = \langle i, Num\rangle, i \in \mathbb{N}^{a}, \\ \langle i, C_i\rangle, & M = \langle N, C_i\rangle^{b}, \\ \langle M, C_{i+j}\rangle, & M = \langle plus\,\langle M', C_i\rangle\,\langle M'', C_j\rangle\rangle^{c}. \end{cases}$$

Intuitively, what we want to happen is:

1. Number literals are marked as constants (*manipulate term*):

$$\langle\langle\langle plus, Num \to Num \to Num\rangle\,\langle x, Num\rangle, Num \to Num\rangle$$
$$\langle\langle\langle plus, Num \to Num \to Num\rangle\,\langle 3, C_3\rangle, Num \to Num\rangle\,\langle 5, C_5\rangle, Num\rangle, Num\rangle$$

2. Propagate type information into arguments (*propagate types*):

$$\langle\langle\langle plus, Num \to Num \to Num\rangle\,\langle x, Num\rangle, Num \to Num\rangle$$
$$\langle\langle\langle plus, C_3 \to C_5 \to Num\rangle\,\langle 3, C_3\rangle, C_5 \to Num\rangle\,\langle 5, C_5\rangle, Num\rangle, Num\rangle$$

3. Apply "sum of constants" rule (*manipulate term*):

$$\langle\langle\langle plus, Num \to Num \to Num\rangle\,\langle x, Num\rangle, Num \to Num\rangle$$
$$\langle\langle\langle plus, C_3 \to C_5 \to C_8\rangle\,\langle 3, C_3\rangle, C_5 \to Num\rangle\,\langle 5, C_5\rangle, Num\rangle, Num\rangle$$

4. Propagate types again:

$$\langle\langle\langle plus, Num \to C_8 \to Num\rangle\,\langle x, Num\rangle, C_8 \to Num\rangle$$
$$\langle\langle\langle plus, C_3 \to C_5 \to C_8\rangle\,\langle 3, C_3\rangle, C_5 \to C_8\rangle\,\langle 5, C_5\rangle, C_8\rangle, Num\rangle$$

5. Prune constant subterms:

$$\langle\langle\langle plus, Num \to C_8 \to Num\rangle\,\langle x, Num\rangle, C_8 \to Num\rangle\,\langle 8, C_8\rangle, Num\rangle$$

We can now strip the extra type information, and get the following optimised term:

$$plus\ x\ 8$$

This section will formalise this process.

[a]mark literals as Constant
[b]substitute terms marked as Constant with a literal
[c]sum with Constant type is of Constant type

The main goal is to be able to tag a term, perform rewriting passes that depend on type information, then propagate any new type information that has arisen after rewriting, repeating the process until[12] it no longer produces a different term, and then untagging back into a regular λ-term. This process is illustrated in fig. 2.1 and is utilised within section 3.3.1 and section 3.3.2.

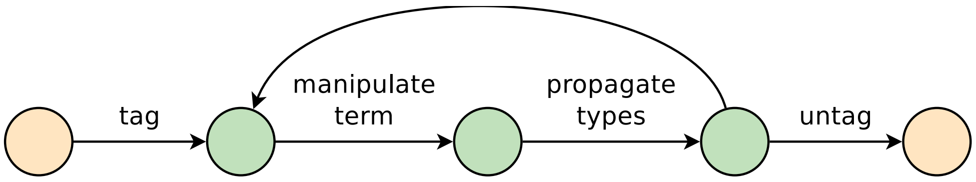


Figure 2.1: Illustration of a term manipulation process. Orange nodes represent typed λ-terms, while green nodes represent type-tagged λ-terms.

We will informally assume that term manipulations do not alter semantics[13] (in a very broad sense).

This "type-tagged term" construction does not need to include casts, since we are storing a type for every subterm anyway, which can replace the use of casts for all practical[14] cases. There is also no need for storing the types of bound variables, since they can be easily extracted from the types of the respective abstraction terms.

Definition 2.46. Let $C \subset$ Const be a set of constants, typed in T.

The set of type-tagged λ-terms Λ_{tT}^{C} is defined inductively[15]:

- Constant:
$$c \in C, \tau \in \mathcal{T}(T) \implies \langle c, \tau \rangle \in \Lambda_{tT}^{C}$$
- Variable:
$$v \in \mathbb{V}^{\lambda}, \tau \in \mathcal{T}(T) \implies \langle v, \tau \rangle \in \Lambda_{tT}^{C}$$
- Application:
$$A, B \in \Lambda_{tT}^{C}, \tau \in \mathcal{T}(T) \implies \langle (AB), \tau \rangle \in \Lambda_{tT}^{C}$$

[12]We will prove that type propagation always terminates. The termination of rewriting, however, depends on the nature of the rewriter itself, and will just be discussed briefly.
[13]by much
[14]There is no way to preserve multiple consecutive casts.
[15]Here we use $\langle M, \tau \rangle$ instead of M^{τ} to denote type tags in order not to confuse them with Church-style typing, which would imply that types are always correct (which we do not require here).

- Abstraction:

$$v \in \mathbb{V}^\lambda, A \in \Lambda^C_{tT}, \tau = (\tau' \to \tau'') \in \mathcal{T}(T), \implies \langle(\lambda v \Rightarrow A), \tau\rangle \in \Lambda^C_{tT}$$

This construction is completely equivalent to the regular typed λ-term construction (type-tags may be emulated by inserting casts at all subterms), and is used only for convenience: this way we can manipulate types directly instead of having to deal with casts and bound variable types.

For convenience, we will also define the notion of *length* on type-tagged terms:

Definition 2.47.

$$|\langle A, \tau\rangle| \overset{\text{def}}{=} \begin{cases} 1, & A = c \in C, \\ 1, & A = v \in \mathbb{V}^\lambda, \\ |P| + |Q| + 1, & A = (PQ), \\ |P| + 1, & A = (\lambda v \Rightarrow P). \end{cases}$$

Converting regular λ-terms into type-tagged λ-terms is relatively straightforward: simply examine the type of every subterm according to its local context and then store it into the tag.

Note that all functions that follow are partial since they are not defined for valid but inadequately typed terms. In implementations, encountering an "undefined" case in any of them is treated as a type error.

We will use the notation "$\twoheadrightarrow$" for denoting partial functions.

Definition 2.48. For a context Γ, $\mathsf{tag}_\Gamma : \Lambda^C_{\leqslant T} \twoheadrightarrow \Lambda^C_{tT}$ is defined inductively:

$$\mathsf{tag}_\Gamma(A) = \begin{cases} \langle c, \|C\|\rangle, & A = c \in C, \\ \langle v, \tau\rangle, & A = v \in \mathbb{V}^\lambda, \Gamma \vdash v : \tau, \\ \langle(\mathsf{tag}_\Gamma(M)\ \mathsf{tag}_\Gamma(N)), \tau\rangle, & A = (MN), \Gamma \vdash A : \tau, \\ \langle\lambda v \Rightarrow \mathsf{tag}_{\Gamma'}(M), \sigma \to \tau\rangle, & A = (\lambda v : \sigma \Rightarrow M), \Gamma' \vdash M : \tau, \Gamma' = \Gamma \circ v : \sigma, \\ \langle M, \tau\rangle, & A = \tau[M]. \end{cases}$$

Although terms returned by tag_Γ are well-typed (the unique type of each subterm according to Γ matches its tag), we want to manipulate those terms before untagging them, and such manipulations may induce discrepancies between actual types and type tags. Thus, our untag_Γ function will take extra care and insert casts where needed:

Definition 2.49. For a context Γ, $\mathsf{untag}_\Gamma : \Lambda^C_{tT} \twoheadrightarrow \Lambda^C_{\leqslant T}$ and $\mathsf{untag}'_\Gamma : \Lambda^C_{tT} \twoheadrightarrow \Lambda^C_{\leqslant T}$ are defined inductively:

$$\mathsf{untag}'_\Gamma(\langle A, \tau\rangle) = \begin{cases} c, & A = c \in C, \\ v, & A = v \in \mathbb{V}^\lambda, \\ \mathsf{untag}_\Gamma(P)\ \mathsf{untag}_\Gamma(Q), & A = (PQ), \\ \lambda v : \tau' \Rightarrow \mathsf{untag}_{\Gamma'}(P), & A = (\lambda v \Rightarrow P), \tau = (\tau' \to \tau''), \Gamma' = \Gamma \circ v : \tau'. \end{cases}$$

$$\mathsf{untag}_\Gamma(\langle A, \tau\rangle) = \begin{cases} M, & M = \mathsf{untag}'_\Gamma(\langle A, \tau\rangle), \Gamma \vdash M : \tau, \\ \tau[M], & M = \mathsf{untag}'_\Gamma(\langle A, \tau\rangle), \Gamma \nvdash M : \tau \text{ and } \tau[M] \text{ is a valid cast}, \\ \neg!, & \text{otherwise.} \end{cases}$$

It can be easily seen that terms returned by untag are well-typed, since we have inserted casts wherever the types do not match. Note that untag may be undefined in some cases (when its definition produces an invalid cast). This means that it was given a term whose type cannot be cast to its type-tag. We will assume that manipulations done to the term will not produce such cases.

Now that we have the construction for attaching types to subterms, we can describe a universal way to propagate information through those types.

We will use a basic operation $\hat{\wedge}$ that unifies two types.

Definition 2.50. An operation $\hat{\wedge} : \mathcal{T}(T) \times \mathcal{T}(T) \twoheadrightarrow \mathcal{T}(T)$ is called a *unifier* for T if:

- it is only defined for types with identical syntactic structure and preserves it, i.e., $\sigma \hat{\wedge} \tau = \nu$ implies that σ, τ and ν have the same syntactic structure[16].
- it is symmetric[17]:
$$\sigma \hat{\wedge} \tau = \nu \implies \tau \hat{\wedge} \sigma = \nu$$
- it is idempotent:
$$\forall \sigma \in \mathcal{T}(T)(\sigma \hat{\wedge} \sigma = \sigma)$$

Definition 2.51. A unifier $\hat{\wedge} : \mathcal{T}(T) \times \mathcal{T}(T) \twoheadrightarrow \mathcal{T}(T)$ is called *monotonic* with respect to the partial order $\leqslant \subseteq \mathcal{T}(T) \times \mathcal{T}(T)$ when:

$$\forall \sigma, \tau \in \mathcal{T}(T)((\sigma \hat{\wedge} \tau \leqslant \sigma) \,\&\, (\sigma \hat{\wedge} \tau \leqslant \tau))$$

A great example for a unifier is the type inference unifier defined in section 3.3.1.

It can also be extended for contexts in the following way:

$$\begin{aligned} \Gamma' \hat{\wedge} \Gamma'' \overset{\text{def}}{=} & \{(x : \sigma \hat{\wedge} \tau) \mid (x : \sigma) \in \Gamma', (x : \tau) \in \Gamma''\} \\ & \cup \{(x : \sigma) \mid (x : \sigma) \in (\Gamma' \cup \Gamma''), (x : \cdot) \notin \Gamma', (x : \cdot) \notin \Gamma''\}. \end{aligned}$$

Now, type propagation will be done in two steps: upward (propagating information up the term tree and into the context) and downward (propagating information down the term tree towards the leaves).

[16]See definition 2.3

[17]Everything in this section may also be done for asymmetric unifiers. Since the need for them has not yet arisen, symmetricity has been assumed for simplicity.

Definition 2.52. For a set of type symbols T and a unifier for T $\hat{\wedge}$, the functions $\mathsf{up}_{\hat{\wedge}}$ that maps a type-tagged term into a new type-tagged term and an inferred context for its free variables, and $\mathsf{down}_{\hat{\wedge}}$ that maps a type-tagged term, a desired type and a free variable context into a new type-tagged term, are defined inductively as follows:

$$\mathsf{up}_{\hat{\wedge}}(\langle A,\tau\rangle) = \begin{cases} (\langle c,\tau\rangle, \{\}), & A = c \in C \\ (\langle v,\tau\rangle, \{v:\tau\}), & A = v \in \mathbb{V}^{\lambda} \\ (\langle \lambda x \Rightarrow \langle M,\sigma'\rangle, (\eta' \to \sigma')\hat{\wedge}\tau\rangle, \Gamma'), & A = (\lambda x \Rightarrow P), \tau = (\eta \to \sigma), \\ & \quad (\langle M,\sigma'\rangle, \Gamma) = \mathsf{up}_{\hat{\wedge}}(P), \Gamma' = \Gamma \backslash (x:\eta'), \\ & \quad (x:\eta') \in \Gamma \\ (\langle \lambda x \Rightarrow \langle M,\sigma'\rangle, (\eta \to \sigma')\hat{\wedge}\tau\rangle, \Gamma'), & A = (\lambda x \Rightarrow P), \tau = (\eta \to \sigma), \\ & \quad (\langle M,\sigma'\rangle, \Gamma) = \mathsf{up}_{\hat{\wedge}}(P) \\ & \quad (x:\cdot) \notin \Gamma \\ (\langle P'Q', \sigma''\hat{\wedge}\tau\rangle, \Gamma_P \hat{\wedge} \Gamma_Q), & A = (PQ), (P', \Gamma_P) = \mathsf{up}_{\hat{\wedge}}(P), \\ & \quad (Q', \Gamma_Q) = \mathsf{up}_{\hat{\wedge}}(Q), P' = \langle M, \sigma' \to \sigma''\rangle \end{cases}$$

$$\mathsf{down}_{\hat{\wedge}}(\langle A,\tau\rangle, \tau', \Gamma) = \begin{cases} \langle c, \tau\hat{\wedge}\tau'\rangle, & A = c \in C \\ \langle v, \tau\hat{\wedge}\tau'\hat{\wedge}\tau''\rangle, & A = v \in \mathbb{V}^{\lambda}, (v:\tau'') \in \Gamma \\ \langle \lambda x \Rightarrow P', \tau''\rangle, & A = (\lambda x \Rightarrow P), \tau'' = \tau\hat{\wedge}\tau' \\ & \quad \tau'' = (\sigma \to \eta), P' = \mathsf{down}_{\hat{\wedge}}(P, \eta, \Gamma \circ (x:\sigma)), \\ \langle P'Q', \tau''\rangle, & A = (PQ), P = \langle M, \sigma \to \eta\rangle, Q = \langle N, \sigma'\rangle, \\ & \quad \tau'' = (\tau'\hat{\wedge}\tau), \sigma'' = (\sigma'\hat{\wedge}\sigma) \\ & \quad P' = \mathsf{down}_{\hat{\wedge}}(P, (\sigma' \to \tau''), \Gamma), Q' = \mathsf{down}_{\hat{\wedge}}(Q, \sigma'', \Gamma) \end{cases}$$

For convenience, the function $\text{propagate}_{\hat{\wedge}}$ is defined as the combination of $\mathsf{up}_{\hat{\wedge}}$ and $\mathsf{down}_{\hat{\wedge}}$:

$$\mathsf{propagate}_{\hat{\wedge}}(P) = \mathsf{down}_{\hat{\wedge}}(\langle M,\tau\rangle, \tau, \Gamma)\text{ , where } \mathsf{up}_{\hat{\wedge}}(P) = (\langle M,\tau\rangle, \Gamma).$$

We will now prove that the iterative application of propagate eventually terminates.

Definition 2.53. If T is a set of type symbols and $\leqslant \subseteq \mathcal{T}(T) \times \mathcal{T}(T)$ is a partial order relation on $\mathcal{T}(T)$, the relation $\leqslant^C_T \subseteq \Lambda^C_{tT} \times \Lambda^C_{tT}$ which works on terms is defined inductively as follows:

- Constant:

$$\frac{a, b \in C \qquad \tau \leqslant \sigma}{\langle a,\tau\rangle \leqslant^C_T \langle b,\sigma\rangle}$$

- Variable:

$$\frac{v, w \in \mathbb{V}^{\lambda} \qquad \sigma, \tau \in \mathcal{T}(T) \qquad \tau \leqslant \sigma}{\langle v,\tau\rangle \leqslant^C_T \langle w,\sigma\rangle}$$

- Application:

$$\frac{A_1, A_2, B_1, B_2 \in \Lambda_{tT}^C \qquad A_1 \leqslant_T^C A_2 \qquad B_1 \leqslant_T^C B_2 \qquad \tau \leqslant \sigma}{\langle (A_1 B_1), \tau \rangle \leqslant_T^C \langle (A_2 B_2), \sigma \rangle}$$

- Abstraction:

$$\frac{v, w \in \mathbb{V}^\lambda \qquad A, B \in \Lambda_{tT}^C \qquad A \leqslant_T^C B \qquad \tau \leqslant \sigma}{\langle (\lambda v \Rightarrow A), \tau \rangle \leqslant \langle (\lambda w \Rightarrow B), \sigma \rangle}$$

Lemma 2.6. If $\hat{\wedge}$ is a monotonic unifier for T with a partial order relation $\leqslant$, then for any type-tagged term $P \in \Lambda_{tT}^C$,

$$(P', \Gamma) = \mathsf{up}_{\hat{\wedge}}(P) \implies P' \leqslant_T^C P.$$

Proof. Let $(\langle A', \tau' \rangle, \Gamma) = \mathsf{up}_{\hat{\wedge}}(\langle A, \tau \rangle)$.

We will prove the lemma inductively.

- $A = c \in C$:
$$\langle A', \tau' \rangle = \underbrace{\langle c, \tau \rangle \leqslant_T^C \langle c, \tau \rangle}_{\text{since } \tau \leqslant \tau} = \langle A, \tau \rangle$$

- $A = v \in \mathbb{V}^\lambda$:
$$\langle A', \tau' \rangle = \underbrace{\langle v, \tau \rangle \leqslant_T^C \langle v, \tau \rangle}_{\text{since } \tau \leqslant \tau} = \langle A, \tau \rangle$$

- $A = (\lambda x \Rightarrow P), \tau = (\eta \to \sigma), (\langle M, \sigma' \rangle, \Gamma) = \mathsf{up}_{\hat{\wedge}}(P), \Gamma' = \Gamma \backslash (x : \eta'), (x : \eta') \in \Gamma$: Inductively, we have that $\langle M, \sigma' \rangle \leqslant_T^C P$. We also know that $\tau' = (\eta \to \sigma') \hat{\wedge} \tau \leqslant \tau$. Thus,
$$\langle A', \tau' \rangle = \langle \lambda x \Rightarrow \langle M, \sigma' \rangle, \tau' \rangle \leqslant_T^C \langle (\lambda x \Rightarrow P), \tau \rangle = \langle A, \tau \rangle.$$

- $A = (PQ), (P', \Gamma_P) = \mathsf{up}_{\hat{\wedge}}(P), (Q', \Gamma_Q) = \mathsf{up}_{\hat{\wedge}}(Q), P' = \langle M, \sigma' \to \sigma'' \rangle$

  Inductively, $P' \leqslant_T^C P$ and $Q' \leqslant_T^C Q$. Since also $\sigma'' \hat{\wedge} \tau \leqslant \tau$, we have:

$$\langle A', \tau' \rangle = \langle P'Q', \sigma'' \hat{\wedge} \tau \rangle \leqslant_T^C \langle PQ, \tau \rangle = \langle A, \tau \rangle.$$

□

Lemma 2.7. If $\hat{\wedge}$ is a monotonic unifier for T with a partial order relation $\leqslant$, then for any type-tagged term $P \in \Lambda_{tT}^C$, any type τ and context Γ,

$$\mathsf{down}_{\hat{\wedge}}(P, \tau, \Gamma) \leqslant_T^C P$$

Proof. Let $\langle A', \tau \rangle = \mathsf{down}_{\hat{\wedge}}(\langle A, \tau \rangle, \sigma, \Gamma)$. We will prove the lemma inductively:

- $A = c \in C$:
$$\langle A', \tau' \rangle = \underbrace{\langle c, \tau \hat{\wedge} \sigma \rangle \leqslant_T^C \langle c, \tau \rangle}_{\text{since } (\tau \hat{\wedge} \sigma) \leqslant \tau} = \langle A, \tau \rangle$$

- $A = v \in \mathbb{V}^\lambda, (v : \tau') \in \Gamma$:

$$\langle A', \tau' \rangle = \underbrace{\langle v, \tau \hat{\wedge} \sigma \hat{\wedge} \tau' \rangle \leqslant^C_T \langle v, \tau \rangle}_{\text{since } (\tau \hat{\wedge} \tau' \hat{\wedge} \sigma) \leqslant \tau} = \langle A, \tau \rangle$$

- $A = (\lambda x \Rightarrow P), \tau'' = \tau \hat{\wedge} \sigma, \tau'' = (\sigma \to \eta), P' = \mathsf{down}_{\hat{\wedge}}(P, \eta, \Gamma \circ (x : \sigma))$: Inductively, $P' \leqslant^C_T P$. Furthermore, $(\tau \hat{\wedge} \sigma) \leqslant \tau$. Thus,

$$\langle A', \tau' \rangle = \langle (\lambda x \Rightarrow P'), \tau \hat{\wedge} \sigma \rangle \leqslant^C_T \langle (\lambda x \Rightarrow P), \tau \rangle = \langle A, \tau \rangle$$

- $A = (PQ), P = \langle M, \nu \to \eta \rangle, Q = \langle N, \nu' \rangle, \tau' = (\sigma \hat{\wedge} \tau), \nu'' = (\nu' \hat{\wedge} \nu), P' = \mathsf{down}_{\hat{\wedge}}(P, (\nu' \to \sigma'), \Gamma), Q' = \mathsf{down}_{\hat{\wedge}}(Q, \nu'', \Gamma)$:

  Inductively, $P' \leqslant^C_T P$ and $Q' \leqslant^C_T Q$. Since also $\tau' = \tau \hat{\wedge} \sigma \leqslant \tau$, we have:

$$\langle A', \tau' \rangle = \langle P'Q', \tau' \rangle \leqslant^C_T \langle PQ, \tau \rangle = \langle A, \tau \rangle$$

□

Lemma 2.8. If $\hat{\wedge}$ is a monotonic unifier for T with a partial order relation $\leqslant$, then for any type-tagged term $P \in \Lambda^C_{tT}$,

$$\mathsf{propagate}_{\hat{\wedge}}(P) \leqslant^C_T P$$

Proof. This follows directly from the definition of propagate, lemma 2.6 and lemma 2.7. □

Lemma 2.9. If T is a set of type symbols and $\leqslant \subseteq \mathcal{T}(T) \times \mathcal{T}(T)$ is a partial order relation on $\mathcal{T}(T)$, the relation $\leqslant^C_T$ is also a partial order relation.

Proof. This lemma can be proven by a simple inductive argument. □

Lemma 2.10. If T is a set of type symbols, $\leqslant \subseteq \mathcal{T}(T) \times \mathcal{T}(T)$ is a partial order relation on $\mathcal{T}(T)$ and $\langle M, \sigma \rangle \leqslant^C_T \langle N, \tau \rangle$, then:

$$|M| = |N| \tag{2.13}$$

$$\sigma \leqslant \tau \tag{2.14}$$

Proof. eq. (2.14) stems trivially from the definition of $\leqslant^C_T$.

eq. (2.13) can be proven by induction over the same definition, since the terms on the left and right of $\leqslant^C_T$ always have the same syntactic structure. □

Lemma 2.11. If T is a set of type symbols, $\leqslant \subseteq \mathcal{T}(T) \times \mathcal{T}(T)$ is a *well-founded* partial order relation on $\mathcal{T}(T)$ and $n \in \mathbb{N}$, then every nonempty subset of Λ^C_{tT} of type-tagged terms of length n has a minimal element with respect to $\leqslant^C_T$.

Proof. We will prove this lemma by strong induction over the term length n.

First, for $n = 1$, we have that any nonempty subset of Λ^C_{tT} contains only Constants or Variables. Each constant or variable is only related to itself, so the proof is trivial.

Now, we will prove the lemma for $n > 1$:

Let $S \subseteq \Lambda^C_{tT}$ be a nonempty set of terms of length n.

Furthermore, let τ_0 be a minimal element of the set $\{\tau \mid \langle M, \tau\rangle \in S\}$ with respect to $\leqslant$.

We know that if the set $S' = \{\langle M, \tau\rangle \mid \tau = \tau_0\}$ has a minimum, it is also the minimum of S, since the contrary would contradict the choice of τ_0 due to eq. (2.14).

Suppose that S' has no minimum, i.e. there exists an infinite strictly-decreasing chain:

$$\ldots <^C_T \langle M_3, \tau_0\rangle <^C_T \langle M_2, \tau_0\rangle <^C_T \langle M_1, \tau_0\rangle <^C_T \langle M_0, \tau_0\rangle$$

such that for every $i \in \mathbb{N}$, $|\langle M_i, \tau_0\rangle| = n$.

Furthermore, the definition of $\leqslant^C_T$ implies that all terms in the chain have the same syntactic structure:

- For every $i \in \mathbb{N}$, M_i is an Application, i.e. $M_i = P_i Q_i$:

  We know that $|P_i| \leqslant |M_i|$ and $|Q_i| \leqslant |M_i|$ since $|M_i| = |P_i| + |Q_i| + 1$.

  Inductively, the sets $\{P_i \mid i \in \mathbb{N}\}$ and $\{Q_j \mid j \in \mathbb{N}\}$ have minimal elements P_i and Q_j for some fixed i and j. Without loss of generality, let $i \geqslant j$.

  We have $\langle M_{i+1}, \tau_0\rangle \leqslant^C_T \langle M_i, \tau_0\rangle$, i.e. $\langle P_{i+1}Q_{i+1}, \tau_0\rangle \leqslant^C_T \langle P_i Q_i, \tau_0\rangle$. The definition of $\leqslant^C_T$ implies that $\langle P_{i+1}, \tau_0\rangle \leqslant^C_T \langle P_i, \tau_0\rangle$: thus, since P_i is minimal, $P_{i+1} = P_i$. Similarly, $Q_{j+1} = Q_j = Q_i = Q_{i+1}$.

  Therefore, the elements $\langle P_{i+1}Q_{i+1}, \tau_0\rangle$ and $\langle P_i Q_i, \tau_0\rangle$ are equal, which contradicts our supposition.

- For every $i \in \mathbb{N}$, M_i is an Abstraction:

  This case can be proven analogously to the case with Application.

□

Lemma 2.12. If T is a set of type symbols and $\leqslant \subseteq \mathcal{T}(T) \times \mathcal{T}(T)$ is a *well-founded* partial order relation on $\mathcal{T}(T)$, the relation $\leqslant^C_T$ is also a well-founded partial order.

Proof. The fact that $\leqslant^C_T$ is a partial order is given by lemma 2.9.

To prove well-foundedness, take a nonempty set $S \subseteq \Lambda^C_{tT}$ and fix $M \in S$. Let $n = |M|$. We can isolate $S_n = \{N \mid |N| = n\}$. By lemma 2.11, S_n has a minimal element. By eq. (2.13) we know that this minimal element is not related to any element in $S \backslash S_n$, therefore it is also minimal in S. □

Proposition 2.13. If $\hat{\wedge}$ is a monotonic unifier for T with a well-founded partial order relation $\leqslant$, then propagate$_{\hat{\wedge}}$ reaches a fixed point on every term, i.e.

$$\forall P \in \Lambda^C_{tT} \quad \exists n \in \mathbb{N} \quad (\text{propagate}^n_{\hat{\wedge}}(P) = \text{propagate}^{n+1}_{\hat{\wedge}}(P))$$

Proof. By lemma 2.8 the sequence $\{A_i\} = \{\text{propagate}^n_{\hat{\wedge}}\}_{n=1}^{\infty}$ is a decreasing chain with respect to $\leqslant^C_T$.

By lemma 2.12 the relation $\leqslant^C_T$ is well-founded.

Then, $\{A_i\}$ has a minimal element A_n for a fixed n. Since $A_{n+1} \leqslant^C_T A_n$ and A_n is minimal, it follows that $A_{n+1} = A_n$. □

In practice, the propagate step shall be combined with a step which manipulates the term for optimisation purposes. This is illustrated in section 3.3.2.

3 Implementation

This section will discuss a practical implementation of an English-to-Overpass translator, written as part of this thesis. The implementation contains a parser for a custom language for defining CCGs, a CCG parser, an intermediate language which can be easily generated via CCGs and translated into Overpass, the respective translator, and a web interface for evaluating queries.

3.1 Architecture

The system consists of several parts:

- Generic CCG definition language: a language for defining CCGs, their respective type systems and term libraries. Can be used standalone (for specifying CCGs that output raw parse trees) or by embedding a lambda calculus-based language (for specifying CCGs that output lambda terms)
- CCG Rule Matcher: a module that tags tokens from the input stream with categories based on the CCG rules (defined in the CCG definition language)
- CCG Parser: a simple proof-of-concept parser based on the CYK algorithm
- English Lexer: a tokeniser, POS-tagger, and lemmatiser for English (largely consists of external components)
- The Minipass language: a small, lambda calculus-based language that translates to Overpass, designed to be easily generated and to represent Overpass operators as operations on a graph.
- Minipass to Overpass translator: a translator from Minipass terms to Overpass queries that does some basic optimisations

An architecture overview can be seen at fig. 3.1.

It is implemented in Haskell, and in addition to the query engine contains also a demonstration web interface through which the user can enter a text query, see the resulting Overpass query, inspect the intermediate steps (derivation tree, minipass term before and after reduction) and finally view the results on a map via Overpass Turbo (a public web interface for visualising queries [wik19c]).

An "*expert*" can write grammar definitions that specify rules for matching tokens and attaching categories and Minipass terms to them. This is done in `.ccg` files, which can then be loaded at runtime and used for parsing the input queries.

A sample grammar definition is provided, which can parse several sentence constructions.

The pipeline can be easily modified to use a natural language other than English, since the only language-dependent components are the tokeniser and the grammar definition, which are straightforward to replace.

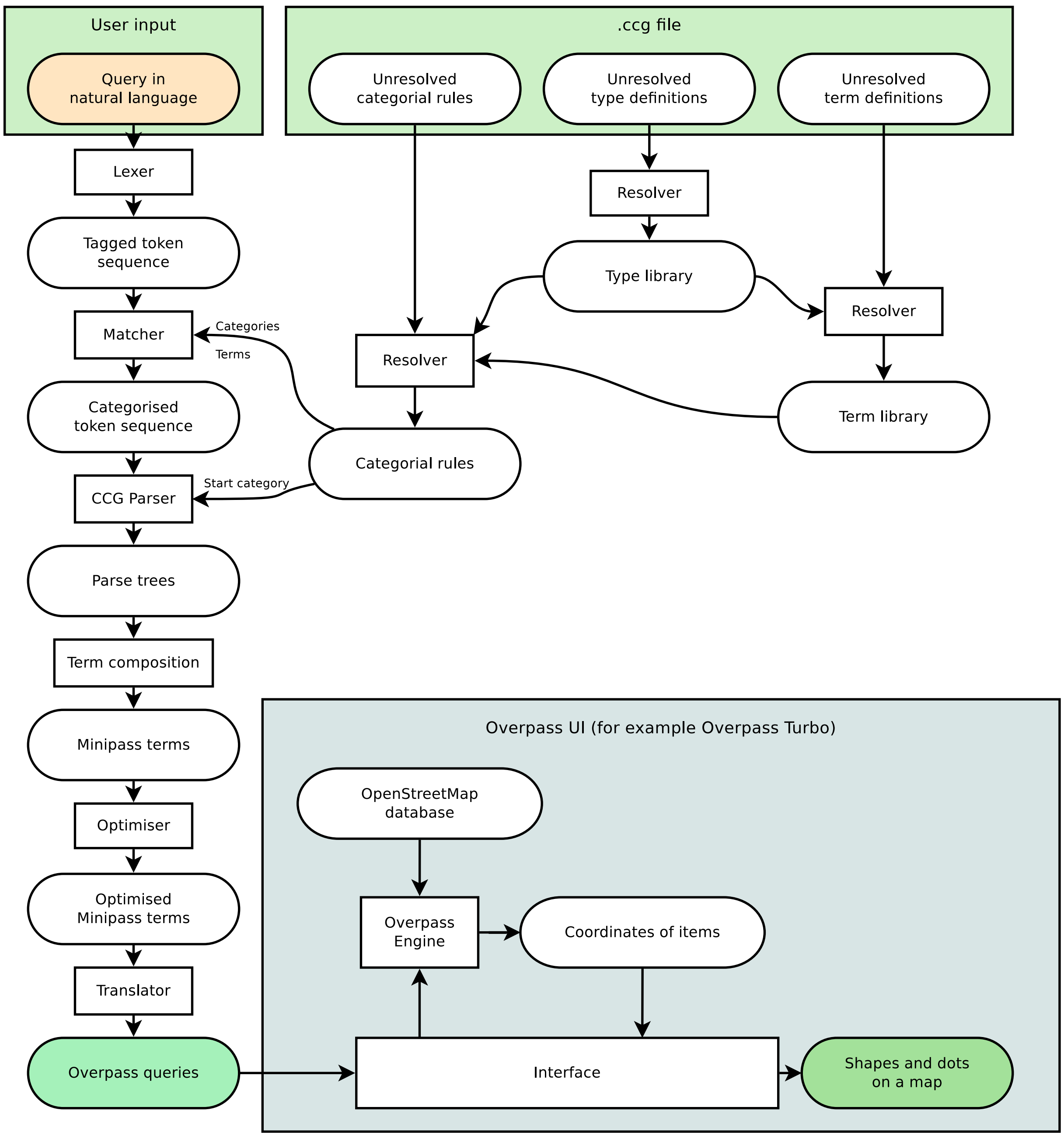


Figure 3.1: Architecture overview

3.2 The Intermediate Language (Minipass)

Overpass is a relatively complex language with assignment, sequentiality, and specialised syntax. Because of this it is not well-suited for performing AST transformations.

Minipass addresses this problem: its goal is to be easy to manipulate and reason about, and be translatable to Overpass. It is based on the typed lambda calculus variant described in section 2.5 and speaks about a world much simpler than the one described by Overpass — a directed graph with labelled edges and vertices.

The graph abstraction works as follows:

- Overpass objects (nodes, ways, relations, areas) are regarded as vertices
- Object tags are encoded into vertex labels: for example, we may have a label for "named London" or "is a cafe".
- Edges labels represent relations between objects: for example, we may have an edge that means "nearby" or an edge "on the same transport route"

Example 3.1. To model a query like `"bus stops near schools in Russia"` using the graph abstraction, we could think like this:

- Let B be the set of all vertices whose label unifies with `"is a bus stop"`.
- Let S be the set of all vertices whose label unifies with `"is a school"`.
- Let R be the set of all vertices whose label unifies with `"is called Russia"`.
- Let N be the set of all vertices we can reach from S by traversing an edge whose label unifies with `"near to"`.
- Let P be the set of all vertices we can reach from R by traversing an edge whose label unifies with `"is inside of"`.
- Our result is $B \cap N \cap P$.

3.2.1 Language grammar

Here is a BNF grammar for the intermediate language.

$$
\begin{array}{lcl}
\langle \textit{program} \rangle & ::= & \langle \textit{tstatement} \rangle \, \langle \textit{tstatement} \rangle^* \\
\langle \textit{tstatement} \rangle & ::= & \langle \textit{statement} \rangle \, \text{‘}\texttt{.}\text{’} \\
\langle \textit{statement} \rangle & ::= & \langle \textit{import-statement} \rangle \\
 & | & \langle \textit{term-definition} \rangle \\
 & | & \langle \textit{type-definition} \rangle
\end{array}
$$

⟨*import-statement*⟩	::=	‘`import`’ ⟨*filepath*⟩
⟨*term-definition*⟩	::=	⟨*identifier*⟩ ‘`:=`’ ⟨*term*⟩
⟨*type-definition*⟩	::=	⟨*identifier*⟩ ‘`<`’ ⟨*type*⟩
⟨*type*⟩	::=	⟨*simple-type*⟩ | ⟨*simple-type*⟩ ⟨*arrow*⟩ ⟨*type*⟩
⟨*simple-type*⟩	::=	⟨*identifier*⟩ | ‘`*`’ | ‘`(`’ ⟨*type*⟩ ‘`)`’
⟨*arrow*⟩	::=	‘`->`’
⟨*term*⟩	::=	⟨*nonapp-term*⟩ | ⟨*nonapp-term*⟩ ⟨*term*⟩
⟨*nonapp-term*⟩	::=	⟨*constant*⟩ | ⟨*variable*⟩ | ⟨*constr*⟩ | ⟨*abstr*⟩ | ‘`(`’ ⟨*term*⟩ ‘`)`’
⟨*constant*⟩	::=	⟨*identifier*⟩ | ⟨*literal*⟩
⟨*variable*⟩	::=	⟨*identifier*⟩
⟨*constr*⟩	::=	⟨*type*⟩ ‘`[`’ ⟨*term*⟩ ‘`]`’
⟨*abstr*⟩	::=	⟨*lambda*⟩ ⟨*declaration*⟩ ‘`=>`’ ⟨*term*⟩
⟨*lambda*⟩	::=	‘`\`’ | ‘`lambda`’
⟨*declaration*⟩	::=	⟨*variable*⟩ | ⟨*variable*⟩ ‘`:`’ ⟨*type*⟩
⟨*literal*⟩	::=	⟨*string-literal*⟩ | ⟨*num-literal*⟩
⟨*string-literal*⟩	::=	⟨*single-quoted-string*⟩
⟨*number-literal*⟩	::=	⟨*decimal-number*⟩
⟨*filepath*⟩	::=	⟨*double-quoted-string*⟩

The definitions for ⟨*single-quoted-string*⟩, ⟨*double-quoted-string*⟩ and ⟨*number-literal*⟩ have been omitted: the first two are plain quoted strings with backslash escaping, while the third is a standard floating point number.

Note that Minipass is characterised by its predefined constants and base types. Because of the modular architecture of the implementation, they can be replaced by something else, allowing to target a different intermediate language.

Applications are left-associative while types are right-associative. Single-line comments can be entered by using the `--` syntax (everything on the same line after the two dashes is ignored by the parser.

Import statements are used to split a program into multiple files, and are resolved relative to the current program's path (or the current working directory when not applicable).

3.2.2 Builtins and base types

Since Minipass is built on top of λ-calculus syntax, all built-in "functions" are actually *constants* in the context of λ-terms.

The base types are:

`Num`	represents a floating point number.
`String`	represents a string of unicode characters.
`List`	represents a polymorphic linked list that may contain `Num`s, `String`s, and nested `List`s.
`GSet`	refers to a set of geographical objects. It is not represented internally in any way: the only way to manipulate `GSet` objects is via builtin operations that work on `GSet`s.

The implementation also allows for the special type * that acts as a wildcard. Typeless declarations are assumed to bear the wildcard type, which is removed from all terms by performing type inference (see section 3.3.1) before translation.

The builtin identifiers (which act as λ-constants) are the following:

Identifier	Type	
`and`	$GSet \to GSet \to GSet$	set intersection
`or`	$GSet \to GSet \to GSet$	set union
`not`	$GSet \to GSet$	set negation
`get`	$List \to GSet$	get geographical elements by vertex label
`next`	$List \to GSet \to GSet$	traverse edges by label, obtaining a new `GSet`
`empty`	$List$	empty list
`consNum`	$Num \to List \to List$	prepend a `Num` to a list
`consString`	$String \to List \to List$	prepend a `String` to a list
`consList`	$List \to List \to List$	prepend a `List` to a list (prepends the list itself, not its elements)

Should the need arise, the addition of extra operations (for example arithmetical) is trivial.

3.2.3 Composing terms

The syntax for composing terms is analogous to the λ-calculus syntax (section 2.5). The valid term constructions are the following:

- Constant: number literal, string literal, or builtin identifier
- Variable: any identifier that is not a builtin
- Application: term application is represented by listing terms separated by whitespace. It is by default left-associative and application order may be adjusted by using round braces.

- Abstraction:

$$\texttt{\textbackslash}\langle \textit{variable}\rangle\texttt{:}\langle \textit{type}\rangle\ \texttt{=>}\ \langle \textit{term}\rangle.$$
or
$$\texttt{\textbackslash}\langle \textit{variable}\rangle\ \texttt{=>}\ \langle \textit{term}\rangle.$$

- Construction:

$$\langle \textit{type}\rangle\texttt{[}\langle \textit{term}\rangle\texttt{]}.$$

3.2.4 Labels and graph queries

The main use for `Lists` is as *labels*: all vertices and edges can be thought to be labelled with a (possibly infinite) set of such labels. Another way to think about labels is as *queries* that retrieve data (either new data in the case of vertex labels, or relational data in the case of edge labels).

For the sake of simplicity the implemented parser does not support a list literal shorthand, but for the following examples we will adopt a list syntax such that

```
[[42, 'bar'], 'foo']
```

means

```
consList
    (consNum 42 (consString 'bar' empty))
    (consString 'foo' empty)
```

Below are the two edge label templates that are currently implemented in the translator:

Label	Semantics
`['around', [<dist>]]`	Items that are within `<dist>` metres from given items
`['in']`	Items that are *inside* the given items

And the three vertex label templates:

Label	Semantics

`['tagFilter',`
` ['=', <key>, <value>]]`
	Items that contain a tag that has the given key and value

`['tagFilter',`
` ['  ', <key>, <regex>]]`
	Items that contain a tag that has the given key and whose value matches the given regular expression

`['all', [<type>]]`	All items in the universe that are of the given Overpass type (way, relation, area, node)

Example 3.2. Here is a query that retrieves all schools in Bristol:

```
and
    (get (consString 'tagFilter'
            (consList (consString '='
                (consString 'amenity'
                    (consString 'school' empty)))
        empty)))
    (next (consString 'in' empty)
        (get (consString 'tagFilter' (consList
            (consString '='
                (consString 'name'
                    (consString 'Bristol' empty)))
        empty))))
```

Since Minipass queries can get too long and cumbersome for humans, a mechanism for modularisation has been implemented. Similarly to most programming languages, the expert can define named terms and types that can later be composed into other terms and types.

3.2.5 Defining types

New types are defined by subtyping an existing type via a "type definition" (subtype assertion) that has the following syntax:

```
NewType < OldType.
```

where `NewType` is not a basic type that is a supertype of `OldType` and has not been already user-defined.

The subtyping semantics are the same as the ones described in section 2.5: all subtype assertions provided by the expert make up a subtype library, which is checked for validity and used for resolving the types contained in all terms.

3.2.6 Defining terms

New terms are defined by using the "term definition" syntax:

```
term_name := some term.
```

Named terms are resolved after parsing by performing free variable substitution in the source code and are bundled in a "term library". The resolver also handles possible type errors and loops.

Note that this language is not Turing complete (not even primitively recursive) since recursion is not handled[18].

Note that *only closed terms* are permitted in definition statements (free variables make little sense).

3.2.7 A "standard" library

This is a small library of term definitions that makes some basic queries easier to write:

```
-- identity
id         := \y => y.

-- the set of all cities
city       := or (or
        (kv 'admin_level' '8')
        (kv 'admin_level' '6'))
    (kv 'capital' '4').

-- within 100 metres
near       :=  within 100.

-- find items by name
name       :=  \x => or (or
        (kv 'name' x)
        (kv 'int_name' x))
    (kv 'name:en' x).

-- find items by name infix
nameLike   :=  \x => or (or
        (contains 'name' x)
        (contains 'int_name' x))
    (contains 'name:en' x).

-- find items by amenity key (fountain, school...)
amenity    :=  kv 'amenity'.
```

[18] Actually, recursion is easy to implement (and was implemented at some point) by adding a recursion combinator and resolving term definition loops to constructions that use that combinator. However, so far the author has not found any use for recursion in such a language apart from abuse.

```
-- find items by tag key and value
kv        :=  \k: String, n: String => get
    (consString 'tagFilter'
        (consList (consString '='
            (consString k (consString n empty)))
    empty)).

-- find items by tag key and value infix
contains  :=  \k: String, n: String => get
    (consString 'tagFilter'
        (consList (consString '~'
            (consString k (consString n empty)))
        empty)).

-- find items within distance of given items
within    :=  \dist : Num => next
    (consString 'around' (consNum dist empty)) .

-- find items inside of given items
in        :=  next (consString 'in' empty).

-- get all items of given type
nodes     :=  get (consString 'all'
    (consString 'nodes' empty)).
ways      :=  get (consString 'all'
    (consString 'ways' empty)).
relations :=  get (consString 'all'
    (consString 'relations' empty)).
areas     :=  get (consString 'all'
    (consString 'areas' empty)).

-- get all items in the universe
everything := get (consString 'all' empty).
```

Example 3.3. With the aid of this term library, the query in example 3.2 becomes more concise:

```
and (amenity 'school') (in (name 'Bristol'))
```

When regarding a standalone Minipass program, the term `main` is used as the query.

3.3 AST Transformations

Minipass terms are subjected to several transformations before emitting an Overpass query. Since type information is used almost everywhere, the term is first converted into a type-tagged term (see definition 2.48) and all transformations (including translation to Overpass) are applied afterwards.

There are three type-systems in use:

- The bare Minipass type system, as described in section 3.2.2
- An expert-defined type system, defined in `.ccg` files by subtyping existing Minipass types (as in section 3.2.5)
- An Intermediate type system which contains more Overpass-centric information and facilitates translation into Overpass, which also subsumes the bare type system

A Minipass term, after being extracted from a CCG derivation tree, is converted to a type-tagged term for easier manipulation and is then subjected to type inference. Afterwards, the expert-defined type information is discarded by squashing the extra types (see proposition 2.12), which gives a regular Minipass type-tagged term, and then setting the type system to the Intermediate type system. This operation does not change the term, since bare Minipass types are also types within the Intermediate type system. Then, several optimisations (some of which take advantage of the extra type information) are performed, and the term is finally translated to Overpass. An illustration of this process is shown at fig. 3.2.

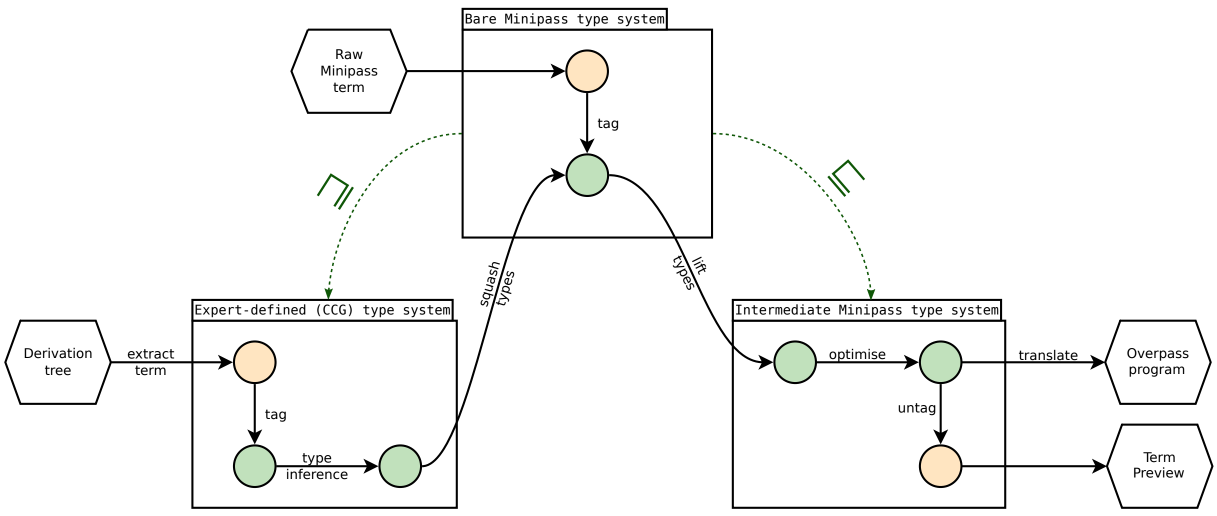


Figure 3.2: Illustration of the transformations undergone by a Minipass term

3.3.1 Type inference

Minipass allows for the usage of the special type $*$ that acts as a wildcard type. It violates the partial order of the type lattice since it is equivalent to all types (i.e. $\forall \sigma \in T : (* \leqslant \sigma) \& (\sigma \leqslant *)$).

Thus, all significant[19] occurrences of $*$ are removed by finding the fixed point of propagate

[19]Since we only work with closed terms, the only case when a type cannot be inferred is when a bound variable is unused (for example in terms like `(\x, y => y) 42`, where the type of `x` cannot be inferred). This, however, is harmless, because the type is unneeded anyway.

from definition 2.52 with the following unifier:

$$\sigma \hat{\wedge} \tau = \begin{cases} \tau, & \sigma = *, \\ \sigma, & \tau = *, \\ \sigma, & \sigma = \tau, \\ (\sigma' \hat{\wedge} \tau') \to (\sigma'' \hat{\wedge} \tau''), & \sigma = (\sigma' \to \sigma''), \tau = (\tau' \to \tau''), \\ \neg!, & \text{otherwise.} \end{cases}$$

Whenever the algorithm encounters a case where the unifier is not defined, a type error is produced.

This unifier is monotonic (definition 2.51) for the partial order "* is above every other type, and there are no other edges". By proposition 2.13, the process of iteratively applying propagate always terminates.

3.3.2 Minipass query optimisation

While Overpass differentiates between 4 types of objects (nodes, ways, relations, areas[20]), its queries are heterogeneous.

Example 3.4. For example, the following Minipass query[a] returns all items with name "Brussels":

```
name 'Brussels'
```

The returned set, among any other objects, contains:

- the city Brussels (a *node*)
- the administrative area of the city Brussels (an *area*)
- the Brussels boulevard in Sofia (a *way*)

Since Overpass contains separate keywords for querying different objects, the above Minipass term would have to be translated to a query as such:

```
( node["name" = "Brussels"];
  way["name" = "Brussels"];
  rel["name" = "Brussels"];
  area["name" = "Brussels"]; ) -> .x1;

.x1 out;
```

In fact, since the current implementation of `name` looks for the name in several fields, the actual emitted Overpass translation is the following:

```
( node["name" = "Brussels"];
  way["name" = "Brussels"];
  rel["name" = "Brussels"];
  area["name" = "Brussels"];

  node["int_name" = "Brussels"];
  way["int_name" = "Brussels"];
  rel["int_name" = "Brussels"];
  area["int_name" = "Brussels"];

  node["name:en" = "Brussels"];
  way["name:en" = "Brussels"];
  rel["name:en" = "Brussels"];
  area["name:en" = "Brussels"]; ) -> .x1;
```

[20]The "area" type is only present in Overpass (not in OpenStreetMap databases), and *area* objects are constructed by finding *relations* that form a closed polygon [wik19a].

```
.x1 out;
```

Now, regard the following Minipass term:

```
in (name 'Brussels')
```

In order to translate it, we could take the result from the previous query and take all nodes, ways or relations[b] inside returned items by using the `area.` filter:

```
( node["name" = "Brussels"];
  way["name" = "Brussels"];
  rel["name" = "Brussels"];
  area["name" = "Brussels"]; ) -> .x1;

( node(area.x1); way(area.x1); rel(area.x1); ) -> .x2;
.x2 out;
```

However, the `area.` filter only takes into account areas from the input set and discards all other objects. Thus, a less naïve translator should only take areas in the first place:

```
( area["name" = "Brussels"]; ) -> .x1;
( node(area.x1); way(area.x1); rel(area.x1); ) -> .x2;
.x2 out;
```

[a]Minipass queries presented here assume the library from section 3.2.7 for clarity.
[b]In Overpass, an *area* may not be inside another *area*

To make this task of optimisation easier, the Intermediate type system extends the $GSet$ type with subtypes for each combination of node, way, relation and area (shown at fig. 3.3): essentially, we attach a set of tags (subset of $\{n, w, r, a\}$) to each $GSet$ type.

During the optimisation phase, several rules are applied to each subterm, some of which are displayed in the table below:[21]:

[21]In this table, "the type of something" means its type tag.

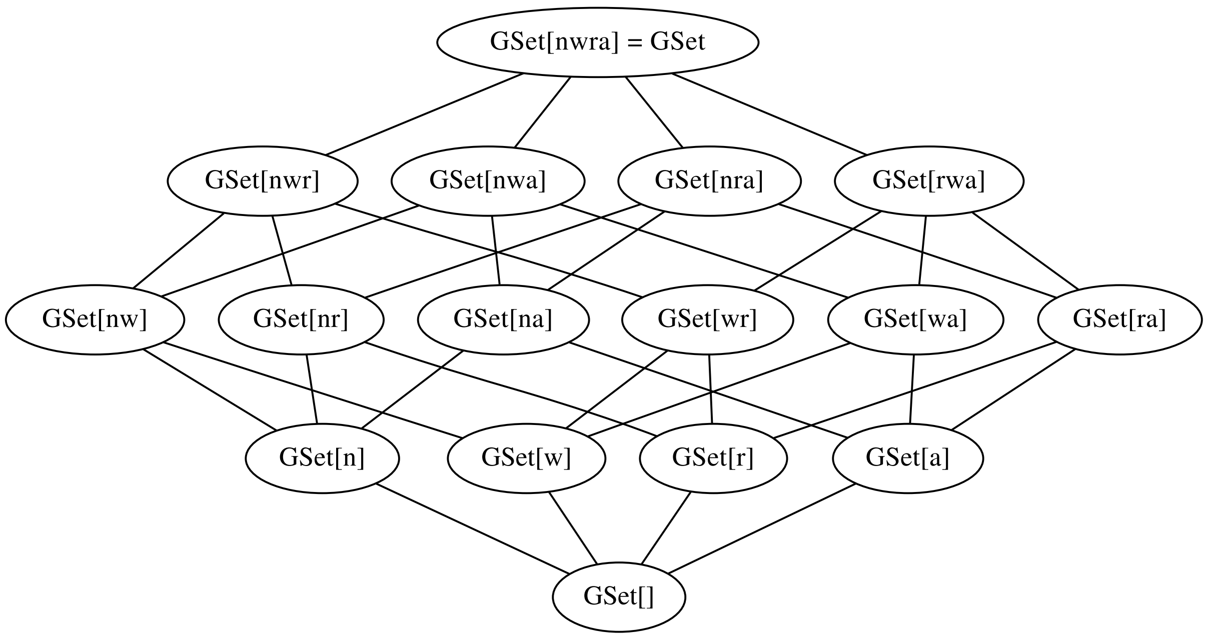


Figure 3.3: The lattice of the $GSet$ type within the Intermediate type system

Subterm	Action
"and" of type $GSet[X] \to GSet[Y] \to GSet[Z]$	set its type to $GSet[X \cap Y \cap Z] \to GSet[X \cap Y \cap Z] \to GSet[X \cap Y \cap Z]$
"or" of type $GSet[X] \to GSet[Y] \to GSet[Z]$	set its type to $GSet[X \cap Z] \to GSet[Y \cap Z] \to GSet[(X \cup Y) \cap Z]$
β-redex	reduce it
"get all nodes" "get all ways" "get all relations" "get all areas"	change it to "get everything" of type $GSet[n]$ change it to "get everything" of type $GSet[w]$ change it to "get everything" of type $GSet[r]$ change it to "get everything" of type $GSet[a]$
"next" of type $GSet[X] \to GSet[Y]$ applied to an "in" label	set its type to $GSet[X \cap \{a\}] \to GSet[Y \cap \{n, w, r\}]$
"or" with one of the arguments being of type $GSet[]$	rewrite the term to only the other argument
...	...

After applying these rules, types are propagated through the term (see definition 2.52) with the following unifier:

$$\sigma \hat{\wedge} \tau \overset{\text{def}}{=} \begin{cases} GSet[X \cap Y], & \sigma = GSet[X], \tau = GSet[Y], \\ Num, & \sigma = Num, \tau = Num, \\ String, & \sigma = String, \tau = String, \\ List, & \sigma = List, \tau = List, \\ (\sigma' \hat{\wedge} \tau') \to (\sigma'' \hat{\wedge} \tau''), & \sigma = (\sigma' \to \sigma''), \tau = (\tau' \to \tau''), \\ \neg!, & \text{otherwise.} \end{cases}$$

We can induce a natural ordering on the finite set of $GSet$ subtypes, so the iterative application of propagate terminates by proposition 2.13.

It can also be seen that the optimisation rules shown above converge after a finite number of steps. A rigorous proof will be omitted[22]

, which is now ready for the translation phase.

While there are many other possible optimisations which can be done here, only these few have been implemented as a proof-of-concept, since this is not the main focus of this work.

3.3.3 Translating to Overpass

The implemented translator can only translate closed terms of non-arrow types. It keeps an internal state which contains the statements generated thus far and the index of the last used variable (used for generating new variable names). At its core is the `translate` function, which takes a Minipass term, returns a `Value` and updates the internal state.

`Value` is defined as follows:

```
data Value
    = StringValue Text
    | NumValue    Float
    | SetValue    VarName
    | ListValue   [ListC]

data ListC
    = NumC        Float
    | StringC     Text
    | ListC       [ListC]

type VarName = Text
```

As can be seen above, values of type $GSet$ (`SetValue`) are represented as a variable name. All other values are represented internally (this means that they are evaluated during translation).

[22] As with the optimisation steps of most compilers, any bugs in the optimiser are left to the user to discover.

Here is an outline of the basic translation process:

- Uncurry[23] the term in order to be able to inspect its arguments
- Recursively evaluate (translate) all arguments (since there are no constants of higher-order type and the term has been fully β-reduced, the head is always a constant and the arguments are of non-arrow types — see property C.1)
- Perform pattern matching of the argument list against a set of translation rules. Modify state and return value according to matched rule (or throw translation error).

Here are some of the aforementioned rules:

#	Input term	Action
1	⟨*string-literal*⟩	return a `StringValue` with the literal's content
2	⟨*number-literal*⟩	return a `NumValue` with the literal's content
3	`ConsString` ⟨*arg1*⟩ ⟨*arg2*⟩	Unpack the `StringValue` returned by the translation of ⟨*arg1*⟩ and prepend it to the `ListValue` returned by the translation of ⟨*arg2*⟩
4	`ConsNum` ⟨*arg1*⟩ ⟨*arg2*⟩	similar to the above
5	`ConsList` ⟨*arg1*⟩ ⟨*arg2*⟩	similar to the above
6	“and”, “or”, “get” or “next”, applied to its respective arguments	Walk the syntax tree of “and”s and “or”s downwards, stopping at leaves (anything that is not “and” or “or”). Create an Overpass filter from the resulting tree. Generate a new variable name and emit an Overpass statement that saves the result from the filter into the variable. Return a `SetValue` bearing said variable name.

At the end of translation, if the result is a `SetValue`, an overpass “`out .<variable>;`” statement is added.

Example 3.5. Consider the following query, generated by the query “pharmacies in Sofia”:

```
and (get (consString 'tagFilter' (consList (consString '='
    (consString 'amenity' (consString 'pharmacy' empty))) empty)))
    (next (consString 'in' empty) (or (or (get (consString
    'tagFilter' (consList (consString '=' (consString 'name'
    (consString 'Sofia' empty))) empty))) (get (consString
    'tagFilter' (consList (consString '=' (consString 'int_name'
    (consString 'Sofia' empty))) empty)))) (get (consString
```

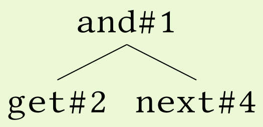
[23] See appendix C

```
    'tagFilter' (consList (consString '=' (consString 'name:en'
    (consString 'Sofia' empty))) empty)))))
```

Since long labels make this code rather unreadable, we will replace them by human-readable placeholders, and will assign numbers to AST nodes (after uncurrying):

```
and                                      -- #1
    (get                                 -- #2
        <amenity=pharmacy>)              -- #3
    (next                                -- #4
        <in>                             -- #5
        (or                              -- #6
           (or                           -- #7
                (get                     -- #8
                    <name=Sofia>)        -- #9
                (get                     -- #10
                    <int_name=Sofia>)    -- #11
            )
            (get                         -- #12
                <name:en=Sofia>)         -- #13
        )
    )
```

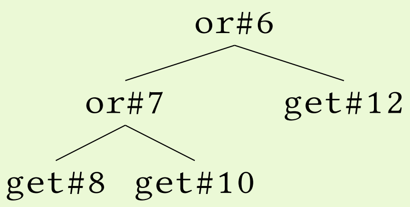
The translator will first apply rule 6 on the following syntax tree:



For this tree, after recursively translating the `amenity#3`, `in#5` and `or#6` subterms, the translator would emit the following Overpass filter:

```
( node(area.<result from or#6>)["amenity" = "pharmacy"]; ) ->
   .<newvar>;
```

So, let's see how `or#6` will be evaluated. Again, we apply rule 6 on the following syntax tree:



After recursively translating `name#9`, `int_name#11` and `name:en#13`, the translator emits this[a] Overpass filter:

```
( area["name" = "Sofia"];
  area["int_name" = "Sofia"];
  area["name:en" = "Sofia"];   ) -> <newvar>;
```

Finally, after substituting the generated variable names and adding an output statement, we get the following query:

```
( area["name" = "Sofia"];
  area["int_name" = "Sofia"];
  area["name:en" = "Sofia"];   ) -> .x1;
( node(area.x1)["amenity" = "pharmacy"]; ) -> .x2;
out .x2;
```

[a]Only "area" nodes are emitted because the type of the "get"s has been inferred as $GSet[a]$ by the optimiser.

While this translator is at proof-of-concept level, it can easily be extended to cover most of Overpass. An overview of features is presented in table 3.1.

Feature	Status
Set Union	Implemented
Set Intersection	Implemented
Set Difference	Trivial to implement
Set Negation	Easy to implement[24]
Equality filter	Implemented
Regex filter	Implemented
Bounding box	Trivial to implement
Recurse (get children)	Trivial to implement
Get item by id	Trivial to implement
Around (within distance)	Implemented
Area (within area)	Implemented
Relation operators (next, previous)	Trivial to implement
Recurse (get children)	Trivial to implement
For-each-loop	Difficult to implement
Complete (fixed point)	Easy to implement
Retro	Not applicable[25]
Counting, summing, stats etc	Difficult to implement
…	…

Table 3.1: Some Overpass features and their corresponding level of support by the translator

3.4 The CCG definition language

When paired with Minipass, the language for defining CCGs consists of the entire Minipass language (section 3.2.1) extended by additional syntax for defining token matchers and categories. It can also be used standalone as a library (in that case, arbitrary payloads can be provided instead of Minipass terms and types).

```
⟨program⟩            ::= ⟨tstatement⟩ ⟨tstatement⟩*
⟨tstatement⟩         ::= ⟨statement⟩ ‘.’
⟨statement⟩          ::= ⟨import-statement⟩
                      |  ⟨begin-statement⟩
                      |  ⟨match-statement⟩
                      |  ⟨term-definition⟩
                      |  ⟨type-definition⟩
⟨import-statement⟩   ::= ‘import’ ⟨filepath⟩
⟨begin-statement⟩    ::= ‘begin’ ⟨category⟩
⟨match-statement⟩    ::= ⟨phrase-rule⟩ ‘:’ ⟨target⟩ (‘,’ ⟨target⟩)*
⟨phrase-rule⟩        ::= ⟨match-rule⟩ (‘<’ ⟨phrase-id⟩ ‘>’ ⟨match-rule⟩)*
⟨match-rule⟩         ::= ⟨matcher⟩ (⟨template⟩ | ε)
⟨matcher⟩            ::= ⟨simple-matcher⟩ | ⟨or-matcher⟩ | ⟨and-matcher⟩
⟨not-or-matcher⟩     ::= ⟨simple-matcher⟩ | ⟨and-matcher⟩
⟨not-and-matcher⟩    ::= ⟨simple-matcher⟩ | ⟨or-matcher⟩
⟨simple-matcher⟩     ::= ⟨exact-matcher⟩ | ⟨regex-matcher⟩ | ‘(’ ⟨matcher⟩ ‘)’
⟨or-matcher⟩         ::= ⟨not-or-matcher⟩ ‘|’ ⟨matcher⟩
⟨and-matcher⟩        ::= ⟨not-and-matcher⟩ ‘&’ ⟨matcher⟩
⟨exact-matcher⟩      ::= ⟨identifier⟩ ‘=’ ⟨double-quoted-string⟩
⟨regex-matcher⟩      ::= ⟨identifier⟩ ‘~’ ⟨double-quoted-string⟩
⟨template⟩           ::= ⟨backtick-quoted-string⟩ ‘:’ ⟨type⟩
⟨target⟩             ::= ⟨category⟩ ‘@’ ⟨term⟩
⟨category⟩           ::= ⟨simple-category⟩ | ⟨simple-category⟩ ⟨slash⟩ ⟨category⟩
⟨simple-category⟩    ::= ⟨type⟩ | ‘(’ ⟨category⟩ ‘)’
⟨slash⟩              ::= ‘/’ ⟨modality⟩ | ‘\’ ⟨modality⟩
⟨modality⟩           ::= ‘.’ | ‘d’ | ‘x’ | ‘*’ | ε
```

Constructions for defining terms, subtyping, importing files, comments and such remain. What the CCG definition language adds is the ability to match natural language tokens and assign categories and terms to them, and the ability to select a starting category.

The starting category is specified with the following statement:

```
begin SomeCategory.
```

There can be only one `begin` statement in the entire program.

3.4.1 Matching tokens

Tokens (words) in the natural language input are emitted by the tokeniser as sets of key-value data (for example, the token `cities` may be represented as `raw=cities,pos=nnp,lemma=city` — for a more detailed explanation see section 3.5.1). The CCG definition language allows us to define *matchers* that can either match a token or not match a token.

The following matchers have been implemented:

- exact matcher (matches tokens that contain a tag with key 'key' and value 'value'):

  ```
  key='value'
  ```

- regex matcher (matches tokens that contain a tag with key 'key' and a value that matches the regular expression 'regex'):

  ```
  key~'regex'
  ```

- "and" matcher (matches tokens that are matched by both given matchers):

  $$\langle \textit{matcher} \rangle \ \& \ \langle \textit{matcher} \rangle$$

- "or" matcher (matches tokens that are matched by any of the given matchers):

  $$\langle \textit{matcher} \rangle \ | \ \langle \textit{matcher} \rangle$$

3.4.2 Assigning categories and terms to matched tokens

The syntax for assigning categories and terms is analogous to the one described in section 2.3.3:

$$\langle \textit{matcher} \rangle : \langle \textit{category} \rangle @ \langle \textit{payload} \rangle$$

where $\langle \textit{payload} \rangle$ is a Minipass term when the language is used to generate Minipass.

Multiple categories and respective terms may be attached to a single matcher by separating them with commas:

$$\langle \textit{matcher} \rangle : \langle \textit{category} \rangle @ \langle \textit{payload} \rangle, \langle \textit{category} \rangle @ \langle \textit{payload} \rangle, \ldots, \langle \textit{category} \rangle @ \langle \textit{payload} \rangle$$

The payload terms can only be closed terms and their type must match the respective category's type.

Categories are constructed as follows:

- Every Minipass type is a valid atomic category (including types defined by the expert via subtyping). Using a complex type for an atomic category is permitted but is seldom useful.

- Complex categories may be constructed by the syntax

$$\langle \textit{category} \rangle \langle \textit{slash} \rangle \langle \textit{category} \rangle$$

  where $\langle \textit{slash} \rangle$ is a modal slash (`/.`, `/s`, `/d` or `/x` that respectively stand for $/_{\cdot}$, $/_{\star}$, $/_{\diamond}$, $/_{\times}$, and their backslash counterparts). The modality may be omitted, and is then assumed to be the dot modality.

- Categorial variables (section 2.4.2) are specified with a dollar sign followed by `[a-zA-Z][a-zA-Z0-9_]*`

Example 3.6. Below is a sample grammar defined using the CCG definition language[a]:

```
import "library.ccg"

-- Use GSet as the starting category.
-- This means that all queries parsed
-- by this CCG will have type GSet.
begin GSet.

-- Here Mult is used to handle lists like
-- "schools, churches, and hospitals"
Mult < GSet.

raw="and"
    : GSet / GSet \ GSet @ or
    , Mult / GSet \ GSet @ \r, l => Mult[or l r].

pos="comma" : $X \ $X @ id -- some commas mean nothing
            , GSet \ GSet / Mult @ or.

raw="in"
    : GSet / GSet \ GSet
    @ \things, where => and things (in where).

lemma="school",pos="nnp" : GSet @ amenity 'school'.
```

[a] Here `"library.ccg"` is assumed to contain the code from section 3.2.7

3.4.3 Composing phrases

In most cases we would would want our rule to match single tokens. Sometimes, however, it is convenient to be able to match an entire phrase (for example, phrases like ‘parking lot’, ‘coworking space’, or ‘rubbish bin’) and assign a category to it.

To achieve this, the CCG definition language supports a syntax for composing phrases (one way to think of it is as "matcher concatenation"):

$$\langle \textit{matcher} \rangle \texttt{ <> } \langle \textit{matcher} \rangle$$
or
$$\langle \textit{matcher} \rangle \texttt{ <DummyCategory> } \langle \textit{matcher} \rangle$$

This phrase syntax is a shorthand for creating a dummy category for each pair and assigning categories such that the entire phrase will be assigned the correct category and term specified for it.

The expert can also explicitly specify the name of the dummy category (this is mainly useful for debugging purposes: the category will show up in the resulting derivation trees).

For more details about this construction, see convention 2.4.

Example 3.7. The following statement:

```
lemma="place" <> raw="of" <> lemma="worship"
    : GSet
    @ amenity 'place_of_worship'.
```

desugars to something like this:

```
Dummy1 < * -> *.
Dummy2 < * -> *.

lemma="place"
    : GSet /s Dummy2 /s Dummy1
    @ \x: Dummy1, y: Dummy2 =>
        amenity 'place_of_worship'.

raw="of"
    : Dummy1
    @ Dummy1[\x => x].

lemma="worship"
    : Dummy2
    @ Dummy2[\x => x].
```

3.4.4 Term templates

Sometimes it is required to insert arbitrary information from tokens into the respective generated terms (e.g. decimal numbers, names, etc.).

This is currently implemented by adding a *template* after the matcher. Templates have the following syntax:

`⟨*template-content*⟩` : ⟨*type*⟩

They are stored in raw form instead of being parsed at the time of parsing the CCG definition source. Apart from regular term syntax, templates may also contain *template variables*, the syntax for which is a dollar sign followed by a valid key that may be found in token data.

Whenever a rule with a template matches a token, template variables are substituted by their values, the template is parsed as a term and is fed as a first argument to the primary term specified in the rule.

Templates are inherently unsafe — failure to parse the template after substitution or having a term with a wrong type produced by the template will cause the CCG parsing process to fail. It is the expert's responsibility to ensure that templates are correct.

Example 3.8. Consider the following rule that contains a template:

```
pos="nnp" `'$raw'` : String
    : GSet
    @ name.
```

It is valid at compile time because the term `name` has type $String \rightarrow GSet$, which yields a term of type $GSet$ after being given a $String$ as argument. When matching the token data "`pos="nnp",raw="Belgium"`", the template `` `'$raw'` `` will become `` `'Belgium'` `` after substitution. Its type (`String`) will be verified, and the resulting term (`name 'Belgium'`) will be constructed.

In the case of multiple concatenated matchers with templates, all templates are fed to the term from left to right. This is achieved by making the generated dummy categories subtypes of the respective template types instead of the identity type. The resulting term then forms naturally.

3.5 Natural language parsing

This section deals with the part of the pipeline which transforms the input query (raw text) into a payload (when targeting Minipass, the payload is a Minipass term).

The process can be broken down into 3 stages:

- Tokenisation: Generating a sequence of tokens from the input text. This stage is natural-language dependant at source-code level (targeting a different natural language requires attaching a different tokeniser). The implementation comes with a simple English tokeniser made with the help of external libraries.
- CCG parsing: Applying the rules specified in the input grammar (which comes as a `.ccg` file) and finding valid parse trees for the input query (if any).
- Term generation: generating final terms by composing the subterms within parse tree leaves

3.5.1 Tokenisation

Tokens are represented as sets of key-value pairs[26]. Currently, there is one implemented tokeniser wrapper, which will be described in this section. Other tokenisers (for different natural languages, for example) may easily be added due to the project's modular architecture.

The English tokeniser wrapper uses a simple tokeniser from the `NLP.POS` package from the `chatter` Haskell library [Cre17]. After splitting the input text into distinct tokens, a POS tagger from the same library is used to assign a part of speech to each token. The POS tagger uses the "averaged perceptron" model [Col02] and is trained on the Penn Treebank POS tag corpus [MMS93].

The raw token string is included under key `raw` in the token object and the POS tag is included under key `pos`.

After POS tagging, tokens are subjected to lemmatisation. This is done using WordNet's `morph` library [Fel98]. Since at the time of writing this thesis available Haskell bindings to WordNet either included no `morph` support or could not run with the most recent version of GHC, a small binding library which wraps the C interface provided with WordNet was written. The `morph` lemmatiser is given a token and its POS tag and returns the lemma for this token. The lemma (if it exists[27]) is then included under key `lemma` in the token object.

The tokeniser also emits special tokens for "begin of sentence" and "end of sentence". These are included as special `mark=begin` and `mark=end` tokens objects.

[26] There is no restriction on the uniqueness of keys.
[27] Tokens like punctuation items have no lemmas.

Example 3.9. The following text:

```
Suddenly there came a tapping.
As if someone gently rapping.
```

Produces the following token list (one token on each row):

```
mark=begin
            raw=Suddenly    pos=rb      lemma=suddenly
            raw=there       pos=ex      lemma=there
            raw=came        pos=vbd     lemma=come
            raw=a           pos=dt      lemma=a
            raw=tapping     pos=nn      lemma=tapping
            raw=.           pos=.       lemma=.
mark=end
mark=begin
            raw=As          pos=in      lemma=as
            raw=if          pos=in      lemma=if
            raw=someone     pos=nn      lemma=someone
            raw=gently      pos=rb      lemma=gently
            raw=rapping     pos=vbg     lemma=rap
            raw=.           pos=.       lemma=.
mark=end
```

3.5.2 CCG parsing

After tokenising, all rule matchers (section 3.4) are evaluated against each token one by one[28] and a list of (Category, Term) pairs is generated for each token. Then, proceeding with the categorial CYK algorithm, a set of items (in the sense of section 2.2.2) is constructed recursively.

The maximum composition arity n in the implementation is set to 1 since no need for more arguments has arisen. If the grammar demands a higher arity, this is easily remedied.

The straightforward recursive implementation of the algorithm relies on a memoiser in order to avoid recomputing shared items. Memoisation is used in several places throughout the code, and two generic memoisers have been implemented:

- A lazy memoiser, based on an infinite bit trie which stores the return value for each possible argument. Only the already accessed paths are stored in memory at a given moment, while the rest of the possible subtrees are represented by Haskell thunks, which are computed on demand.
- A pragmatic memoiser which stores computed values in a hashmap which is accessed via `unsafePerformIO`.

[28]This naïve quadratic approach may be replaced by a decision tree-based traversal of the set of matchers with little effort.

The lazy memoiser, while elegant, is about twice as slow as the pragmatic memoiser, which is used by default.

Another performance consideration is the fact that when combining categories, care is taken to share common parts of categories with ones constructed from them[29].

3.5.3 Parse forests and generating terms

Since there are potentially an exponential number of derivation trees for a single query, many of them share an identical trunk due to the way the categorial CYK algorithm works. Thus, it is beneficial to encode the resulting parse forest in such a way as to share the common trunks between trees, and also be able to avoid enumerating all of them when performing simple manipulations.

The listing below is a simplified definition of the `ParseTree` and `ParseForest` datatypes used in the implementation. Here, `cat` is a category, `payload` is a payload attached to the matched token (a λ-term and metadata about the token itself), and `(CombineRule cat)` is a rule[30] by which this derivation was produced (used for debugging and visualisation):

```
data ParseTree cat payload where
    Leaf cat payload |
    Vert cat (CombineRule cat) (ParseTree cat payload) (ParseTree cat
       payload)

data ParseForest cat payload
    = ParseForest cat [MultiNode cat payload]

data MultiNode cat payload where
    MultiLeaf payload           (MultiNode cat payload) |
    MultiVert (CombineRule cat) (ParseForest cat payload) (ParseForest
       cat payload)
```

A `ParseForest` is recursively generated from the set of items produced by the parsing algorithm, starting from the initial category and the whole query.

From the `ParseForest`, all `ParseTree`s are enumerated in a lazy list: this way most of the encoded trees will never be evaluated individually (unless requested by the user).

A λ-term is generated for each relevant tree, following the procedure from definition 2.22. Afterwards, in the case of Minipass terms, their types are squashed[31] into the bare Minipass type system, discarding any type information that was useful only for parsing.

[29]This comes for free because of Haskell's immutability mechanism: the only care that needs to be taken is not obstructing the compiler in that regard.

[30]The syntax "`(CombineRule cat)`" relies on the `TypeFamilies` Haskell extension. Type family constraints have been omitted for brevity.

[31]See definition 2.40.

4 Building and usage

4.1 Directory layout

The project's source code has the following directory structure:

- `ccg` — CCGs and CCG parsing
- `lambda-calculus` — λ-terms, type systems, and type-tagging
- `utils` — Various utilities
- `thething` — Project entry point (CLI interface)
- `examples`
  - `library.ccg` — A library of useful Minipass terms (see section 3.2.7)
  - `amenities.ccg` — CCG matching rules for the 170 most frequently used OpenStreetMap amenities
  - `sample_minipass.ccg` — a sample Minipass term
  - `sample_grammar.ccg` — a sample CCG which can parse several types of sentences in English

4.2 Running

The CLI can run in 3 modes:

- *summary mode* — given an input query, generate at most five distinct Minipass terms and display them in the terminal along with the respective Overpass translations
- *LaTeX mode* — given an input query, generate a PDF file with at most 10 parses, visualisations of their derivation trees, respective Minipass terms and Overpass translations
- *web interface mode* — start a local web server, which provides an user interface for entering queries, inspecting the results including derivation trees, type hierarchies, and token data, and visualising Overpass results on a map via Overpass Turbo[32].

Note: For using the LaTeX mode and for being to inspect derivation trees within the web interface, `xelatex` and `latexmk` must be installed.

For a build system, Haskell's `Stack` has been used. The CLI can be run by issuing:

```
$ stack run underpass-exe <arguments>
```

[32]Overpass Turbo is a publicly-available Overpass frontend [wik19c]

When run without arguments, the program loads `examples/sample_grammar.ccg` and starts in web interface mode.

Other modes can be used as follows:

```
# Start in web-interface mode and load my_grammar.ccg:
$ stack run underpass-exe serve my_grammar.ccg

# Display summary for a given query:
$ stack run underpass-exe summary my_grammar.ccg 'hospitals in China'

# Generate a PDF breakdown for a given query:
$ stack run underpass-exe latex my_grammar.ccg 'hospitals in China'
```

While the actual running time with the sample grammar is negligible, the CLI must load the trained POS tagger and lemmatiser on startup, which takes several seconds. Also, drawing trees is relatively slow because it uses LaTeX, so some patience is required.

4.3 Sample results

These are the contents of `examples/sample_grammar.ccg`:

```
import "library.ccg".
import "amenities.ccg".

begin S.

SE < GSet.

S < GSet.

Entity < GSet.
NamedEntity < String.

mark="begin"  : S / SE        @ \x: SE     => S[GSet[x]] .
mark="end"    : SE \ GSet     @ \x: GSet   => SE[x] .

pos="comma"   : $X \ $X       @ id.

raw="in"
    : GSet / GSet \ GSet
    @ \things, where => and things (in where).

pos="nnp" `'$raw'` : String
    : Entity
    @ name.
```

```
pos="nnp" `'$raw'` : String
    : NamedEntity
    @ \x => NamedEntity[x].

raw="like"    : GSet \ NamedEntity @ nameLike.
raw="city"    : GSet \ Entity      @ \x : Entity => and city x.
raw="near"    : GSet \ GSet / GSet @ \b, a => and a (near b).
```

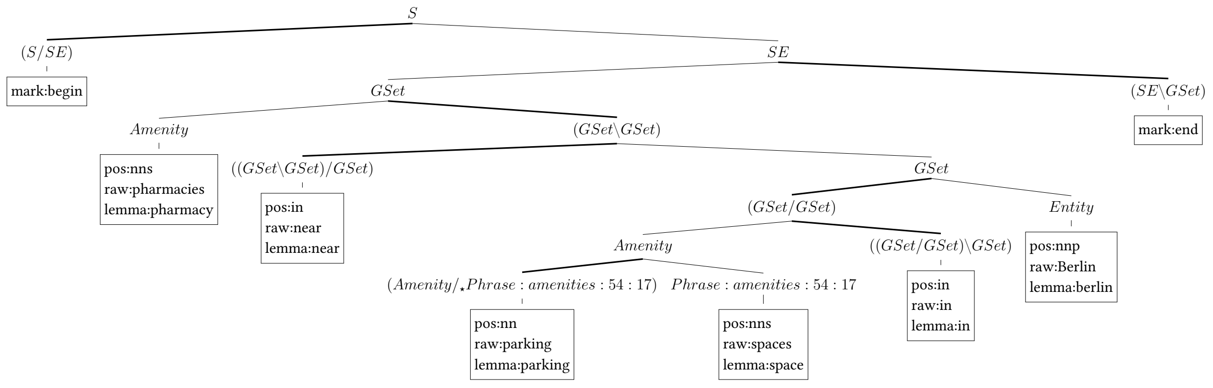
We will now, for a more elaborate example, inspect the following query:

`pharmacies near parking spaces in Berlin`

A relevant excerpt from `amenities.ccg` (which defines the phrase "parking space") is the following:

```
lemma="parking" <> lemma="space"
    : Amenity
    @ amenity 'parking_space'.
```

So, one[33] of the derivation trees generated for this query is given below:


S
(S/SE)
mark:begin
SE
GSet
(SE\GSet)
mark:end
Amenity
pos:nns raw:pharmacies lemma:pharmacy
(GSet\GSet)
((GSet\GSet)/GSet)
pos:in raw:near lemma:near
GSet
(GSet/GSet)
Entity
pos:nnp raw:Berlin lemma:berlin
Amenity
((GSet/GSet)\GSet)
pos:in raw:in lemma:in
(Amenity/⋆Phrase : amenities : 54 : 17)
pos:nn raw:parking lemma:parking
Phrase : amenities : 54 : 17
pos:nns raw:spaces lemma:space

This tree yields the following Minipass term:

```
λ ((λ x ⇒ S[GSet[x]]) ((λ x ⇒ SE[x]) ((λ b ⇒ (λ a ⇒ ((Amenity  →
   (GSet  →  GSet))[and] a ((λ dist ⇒ (next (consString 'around'
   (consNum dist empty)))) 100.0 b)))) ((λ things ⇒ (λ where ⇒
   ((Amenity  →  (GSet  →  GSet))[and] things ((List  →  (Entity  →
   GSet))[next] (consString 'in' empty) where)))) ((λ _ ⇒ ((λ k ⇒ (λ n
   ⇒ ((List  →  Amenity)[get] (consString 'tagFilter' (consList
   (consString '=' (consString k (consString n empty))) empty)))))
   'amenity' 'parking_space')) Phrase:amenities:54:17[(λ _ ⇒ _)]) ((λ
   x ⇒ ((GSet  →  (GSet  →  Entity))[or] (or ((λ k ⇒ (λ n ⇒ (get
   (consString 'tagFilter' (consList (consString '=' (consString k
   (consString n empty))) empty))))) 'name' x) ((λ k ⇒ (λ n ⇒ (get
   (consString 'tagFilter' (consList (consString '=' (consString k
```

[33]We get two semantic parses for this query: "pharmacies near parking spaces, said pharmacies being in Berlin" and "pharmacies near parking spaces, said parking spaces being in Berlin". Both are valid.

```
    (consString n empty))) empty))))) 'int_name' x)) ((λ k ⇒ (λ n ⇒
    (get (consString 'tagFilter' (consList (consString '=' (consString
    k (consString n empty))) empty))))) 'name:en' x))) 'Berlin')) ((λ k
    ⇒ (λ n ⇒ ((List  →  Amenity)[get] (consString 'tagFilter' (consList
    (consString '=' (consString k (consString n empty))) empty)))))
    'amenity' 'pharmacy'))))
```

Whose types are then squashed:

```
(( λ x ⇒ x) (( λ x ⇒ x) (( λ things ⇒ ( λ where ⇒ (and things (next
    (consString 'in' empty) where)))) (( λ b ⇒ ( λ a ⇒ (and a (( λ dist
    ⇒ (next (consString 'around' (consNum dist empty)))) 100.0 b)))) ((
    λ _ ⇒ (( λ k ⇒ ( λ n ⇒ (get (consString 'tagFilter' (consList
    (consString '=' (consString k (consString n empty))) empty)))))
    'amenity' 'parking_space')) ( λ _ ⇒ _)) (( λ k ⇒ ( λ n ⇒ (get
    (consString 'tagFilter' (consList (consString '=' (consString k
    (consString n empty))) empty))))) 'amenity' 'pharmacy')) (( λ x ⇒
    (or (or (( λ k ⇒ ( λ n ⇒ (get (consString 'tagFilter' (consList
    (consString '=' (consString k (consString n empty))) empty)))))
    'name' x) (( λ k ⇒ ( λ n ⇒ (get (consString 'tagFilter' (consList
    (consString '=' (consString k (consString n empty))) empty)))))
    'int_name' x)) (( λ k ⇒ ( λ n ⇒ (get (consString 'tagFilter'
    (consList (consString '=' (consString k (consString n empty)))
    empty))))) 'name:en' x))) 'Berlin'))))
```

And is then β-reduced:

```
(and (and (get (consString 'tagFilter' (consList (consString '='
    (consString 'amenity' (consString 'pharmacy' empty))) empty)))
    (next (consString 'around' (consNum 100.0 empty)) (get (consString
    'tagFilter' (consList (consString '=' (consString 'amenity'
    (consString 'parking_space' empty))) empty))))) (next (consString
    'in' empty) (or (or (get (consString 'tagFilter' (consList
    (consString '=' (consString 'name' (consString 'Berlin' empty)))
    empty))) (get (consString 'tagFilter' (consList (consString '='
    (consString 'int_name' (consString 'Berlin' empty))) empty)))) (get
    (consString 'tagFilter' (consList (consString '=' (consString
    'name:en' (consString 'Berlin' empty))) empty))))))
```

Finally, after optimisation and translation, we get an Overpass query:

```
( node["amenity" = "parking_space"]; ) -> .x1;
( area["name" = "Berlin"]; area["int_name" = "Berlin"]; area["name:en"
    = "Berlin"]; ) -> .x2;
( node(area.x2)(around.x1:100.0)["amenity" = "pharmacy"]; ) -> .x3;
.x3 out;
```

When rendering this query with Overpass Turbo, as of the time of writing this document, we get exactly the result shown in fig. 1.1.

5 Closure

The primary objective of this thesis was to explore the suitability CCGs for formulating geospatial queries in a formal language based on a natural language phrase. English was chosen as a concrete natural language while Overpass was chosen as a target formal language. To achieve this goal, the following tasks were accomplished:

- Design of a functional intermediate language that can be easily generated and translated into Overpass
- Design of a language for defining CCGs on top of aforementioned intermediate language
- Implementation of parsers and internal representations for said languages
- Design, implementation and prooves of correctness for a subtyping system and a type variable system for providing polymorphism
- Implementation and proof of correctness of a Categorial CYK-based CCG parser
- Implementation of an intermediate language term generator from CCG derivation trees
- Implementation and proof of termination of a translator from intermediate language terms into Overpass programs
- Implementation of user interfaces (console and web-based)

In the process it became clear that defining the exact subset of English was not as interesting as expected, so the project grew to allow definition of frontends for any natural language subsets. This shifted the focus into the design of a rule definition language, which felt more fulfilling.

It seems clear that this CCG-based approach can't model the input natural language as well as some methods based on statistics would have (the language subset modelled by CCG feels more formal than actual natural language sentences), but it may prove useful for many scenarios: this "formality" ensures predictability of the system and still models enough of the natural language for queries to be semantically-recognisable to non-experts. Interfacing such a system may be much easier to learn than working directly with a formal language such as Overpass.

In retrospect, the subjectively most interesting part of this thesis was probably incorporating the concept of subtyping into CCGs and using it to model real-world semantics, and also the most tedious, since the details arising from contravariance had to be considered in most prooves.

Even without using Overpass as a target formal language, this project might be useful for generating queries into other formal languages by simply substituting the backend.

5.1 Future work

The current implementation is fairly basic and may be extended along multiple axes:

- Implement more Overpass features into the translator to cover a greater target domain
- Allow assigning weights to CCG rules in order to be able to prioritise different parses for the same query
- Implement a version of the Vijay-Shanker algorithm for CCG parsing in order to improve performance
- Avoid β-reduction on Minipass terms of type $GSet$ in order to eliminate some duplicate subqueries
- Use Eisner normal form parsing to handle ambiguity better
- Use learning methods to automatically extend grammars by using query corpora

5.2 Conclusion

Mostly harmless.

Appendices

A Of mythical creatures and magical sets

Sooner or later, in every work related to formal logic, the need for a set of basic objects arises. For example, we want to construct a system for basic arithmetic, for which we need "variables". At first glance, this is a trivial matter — just take an arbitrary countable set and call it a "set of variables" - but wait! What if the left parenthesis symbol ends up in this set? Well, let's fix that: we banish all parentheses from our set of variables $\mathbb{V}$.

$$\{(,)\} \cap \mathbb{V} = \varnothing$$

Fine. But what if the number 7 ends up in our set of variables? Then, when we say 7, did we mean the number or the variable? Well. No place for numbers in our set of variables as well.

$$(\{(,)\} \cup \mathbb{N}) \cap \mathbb{V} = \varnothing$$

But wait! What if the string (42) ends up in our set of variables? That will surely lead to problems. Try again:

$$(\{(,)\} \cup \mathbb{N})^* \cap \mathbb{V} = \varnothing$$

Alas, this is still not enough. We haven't excluded strings like $foo(bar$.

Hmm, maybe this problem isn't that easy after all — why not give up and make $\mathbb{V}$ be the set of all splotches of ink that an ancient Greek would recognise as a sequence of their letters?

As entertaining as it is to think about this problem instead of trying to write the actual thesis, it's time for a quick solution. One way to wave hands about this is to assume that $\mathbb{V}$ is a magical set that contains no elements that we have used anywhere: essentially a set that is disjoint with any set we feel convenient that it be disjoint with. It contains none of the special symbols we use elsewhere.

Another way is to construct a very specific set, which is what we'll do here:

$$\mathbb{V} \overset{\text{def}}{=} \{\mathrm{v}, \mathrm{v}', \mathrm{v}''...\},$$

where v is the specific symbol v that is used nowhere else.

This construction will be used everywhere we need a set of unique symbols.

B Type variables in λ-terms

This section extends the version of simply-typed λ-calculus from section 2.1 to include *type variables*. The same construction can also be applied to simply-typed λ-calculus with subtypes (section 2.5) without modification.

This construction closely resembles the parametric polymorphism construction from [Pie02, chap. 25].

Let $\mathbb{V}^T \overset{\text{def}}{=} \{\tau', \tau'', \tau''', ...\}$ be a set of symbols we call *type variables*.

First, we will extend definition 2.1 to include variables:

Definition B.1. For a set of type symbols T, its *type closure with type variables* $\mathcal{T}_v(T) \subseteq (T \cup \mathbb{V}^T \cup \{\rightarrow,), (\})^*$ is defined inductively:

- $\alpha \in \mathbb{V}^T \implies \alpha \in \mathcal{T}_v(T)$
- $\sigma \in T \implies \sigma \in \mathcal{T}_v(T)$
- $\sigma, \tau \in \mathcal{T}_v(T) \implies (\sigma \rightarrow \tau) \in \mathcal{T}_v(T)$

Definition B.2. We call a set X *typed with variables in* $T \iff$ there is a function

$$\| \cdot \|_v : X \rightarrow \mathcal{T}_v(T)$$

Now we need type variable substitution:

Definition B.3. For a type $\sigma \in \mathcal{T}_v(T)$, a type variable α and another type $\tau \in \mathcal{T}_v(T)$, the type variable substitution $\sigma[\alpha := \tau]$ is defined recursively as follows:

$$\sigma[\alpha := \tau] \overset{\text{def}}{=} \begin{cases} \sigma, & \sigma \in T, \\ \sigma, & \sigma \in \mathbb{V}^T \backslash \{\alpha\}, \\ \tau, & \sigma = \alpha, \\ \sigma'[\alpha := \tau] \rightarrow \sigma''[\alpha := \tau], & \sigma = \sigma' \rightarrow \sigma''. \end{cases}$$

And now we can define λ-terms (the definition is completely identical to definition 2.6 except for the fact that types may contain variables:

Definition B.4. Let $C \subset \text{Const}$ be a set of constants, typed with variables in T.

Λ_{aT}^C is defined inductively:

- Constant:
$$c \in C \implies c \in \Lambda_{aT}^C$$
- Variable:
$$v \in \mathbb{V}^\lambda \implies v \in \Lambda_{aT}^C$$
- Application:
$$A, B \in \Lambda_{aT}^C \implies (AB) \in \Lambda_{aT}^C$$
- Abstraction:
$$v \in \mathbb{V}^\lambda, A \in \Lambda_{aT}^C, \sigma \in \mathcal{T}_v(T) \implies (\lambda v : \sigma \Rightarrow A) \in \Lambda_{aT}^C$$

The definitions for free variables, declaration, context and judgement shall also be reused from section 2.1.

Now, we will redefine the derivation rules for typed terms (definition 2.9), extending them with an additional rule that lets type variables act as any type:

Definition B.5. Derivation rules for typed λ-terms:

- Constant

$$\frac{c \in C}{\Gamma \vdash c : \|c\|_v}$$

- Variable

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

- Application

$$\frac{\Gamma \vdash A : \sigma' \to \sigma'' \qquad \Gamma \vdash B : \sigma'}{\Gamma \vdash (AB) : \sigma''}$$

- Abstraction

$$\frac{\Gamma \circ x : \tau \vdash A : \sigma}{\Gamma \vdash (\lambda x : \tau \Rightarrow A) : \tau \to \sigma}$$

- Type variable substitution

$$\frac{\Gamma \vdash A : \sigma \qquad \alpha \in \mathbb{V}^T \qquad \tau \in \mathcal{T}_v(T)}{\Gamma \vdash A : \sigma[\alpha := \tau]}$$

Also, for convenience, we will define type variable substitution over all types within a term:

Definition B.6. For $\alpha \in \mathbb{V}^T$, $A \in \Lambda^C_{aT}$ and $\tau \in \mathcal{T}_v(T)$,

$$A[\alpha := \tau] \overset{\text{def}}{=} \begin{cases} c, & A = c \in C, \\ v, & A = v \in \mathbb{V}^\lambda, \\ M[\alpha := \tau]\ N[\alpha := \tau], & A = MN, \\ \lambda x : \sigma[\alpha := \tau] \Rightarrow M[\alpha := \tau], & A = (\lambda x : \sigma \Rightarrow M). \end{cases}$$

Note that while for consistency we allow type variables within the types of constants, this rarely arises in practice. In particular, the current implementation does not use variables in constant types anywhere.

C β-reduction and (un)currying

C.1 β-reduction

This section briefly explains the concept of β-reduction. For more details, see [Pie02].

Definition C.1. For $M, N \in \Lambda_T^C$, the substitution of the free variable x within M with N is defined inductively as follows:

$$M[x := N] \overset{\text{def}}{=} \begin{cases} N, & M = x \\ y, & M = y \in \mathbb{V}^\lambda \backslash \{x\} \\ \lambda x : \sigma \Rightarrow P[x := N], & M = (\lambda x : \sigma \Rightarrow P), y \neq x \; \& \; y \notin \mathsf{fv}(P) \\ A[x := N]B[x := N], & \text{M = (AB)} \end{cases}$$

Note that this function is partial, but this can be swept under the rug by renaming bound variables when needed [Pie02].

We will now define the β-reduction relation $\xrightarrow{\beta}$ between λ-terms:

Definition C.2. Two terms are in $\xrightarrow{\beta}$ relation only in the following case:

$$(\lambda x : \sigma \Rightarrow M)N \xrightarrow{\beta} M[x := N]$$

If $A \xrightarrow{\beta} B$, A is called a β-*redex*.

The relation $\overset{\beta}{\Rightarrow}$ β-reduces any subterm:

Definition C.3. The $\overset{\beta}{\Rightarrow}$ relation is defined inductively:

- $A \xrightarrow{\beta} B \implies A \overset{\beta}{\Rightarrow} B$
- $A \overset{\beta}{\Rightarrow} B \implies AP \overset{\beta}{\Rightarrow} BP$
- $A \overset{\beta}{\Rightarrow} B \implies PA \overset{\beta}{\Rightarrow} PB$
- $A \overset{\beta}{\Rightarrow} B \implies (\lambda x : \sigma \Rightarrow A) \overset{\beta}{\Rightarrow} (\lambda x : \sigma \Rightarrow B)$

Definition C.4. A term $M \in \Lambda_T^C$ is called *fully β-reduced* if:

$$\forall N \in \Lambda_T^C(\neg(M \overset{\beta}{\Rightarrow} N))$$

C.2 Uncurrying

We can represent every λ-term as a function (*head*) applied to a sequence of arguments:

- If the given term M is not an application, its head is M itself with no arguments.
- If the given term is $M = (AB)$, its head is the head of A and its arguments are B appended to the arguments of A.

Example C.1. The term $P(QR)(\lambda x : \sigma \Rightarrow S)T$ has head P and arguments (QR), $(\lambda x : \sigma \Rightarrow S)$ and T.

Property C.1. If a term:

- is closed
- contains no β-redex
- contains no constants of higher-order-type
- is of non-arrow type

then its arguments have non-arrow types.

Proof. Let $M \in \Lambda_T^C$ be a term which satisfies the conditions above.

Since M is of non-arrow type and contains no β-redex, it contains no abstractions, therefore no bound variables. Since it is closed, this means it also contains no free variables.

Thus, M only contains constants and applications, and must have the following form:

$$((...(cM_1)...M_{k-1})M_k)$$

where the head c is a constant and the arguments $M_1...M_k$.

Since c is a constant of first-order type, its arguments have simple types. □

References

[BB11] Robert Borsley and Kersti Börjars. *Non-Transformational Syntax: Formal and Explicit Models of Grammar.* June 2011. DOI: 10.1002/9781444395037.

[BK03] Jason Baldridge and Geert-Jan M. Kruijff. "Multi-modal Combinatory Categorial Grammar". In: *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 1.* EACL '03. Budapest, Hungary: Association for Computational Linguistics, 2003, pp. 211–218. ISBN: 1-333-56789-0. DOI: 10.3115/1067807.1067836. URL: https://doi.org/10.3115/1067807.1067836.

[Col02] Michael Collins. "Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms". In: *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002).* Association for Computational Linguistics, July 2002, pp. 1–8. DOI: 10.3115/1118693.1118694. URL: https://www.aclweb.org/anthology/W02-1001.

[Cre17] Rogan Creswick. *chatter: A library of simple NLP algorithms.* 2017. URL: http://web.archive.org/web/20190716133014/http://hackage.haskell.org/package/chatter-0.9.1.0.

[doc19] Dart language documentation. *Fixing common type problems.* 2019. URL: https://web.archive.org/web/20190602183435/https://dart.dev/guides/language/sound-problems#the-covariant-keyword.

[Eis96] Jason Eisner. "Efficient Normal-form Parsing for Combinatory Categorial Grammar". In: *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics.* ACL '96. Santa Cruz, California: Association for Computational Linguistics, 1996, pp. 79–86. DOI: 10.3115/981863.981874. URL: https://doi.org/10.3115/981863.981874.

[Fel98] Christiane Fellbaum, ed. *WordNet: An Electronic Lexical Database.* Language, Speech, and Communication. Cambridge, MA: MIT Press, 1998. ISBN: 978-0-262-06197-1.

[HB10] Julia Constanze Hockenmaier and Yonatan Bisk. "Normal-form parsing for combinatory categorial grammars with generalized composition and type-raising". English (US). In: *Coling 2010 - 23rd International Conference on Computational Linguistics, Proceedings of the Conference.* Vol. 2. 2010, pp. 465–473.

[KKS10] Marco Kuhlmann, Alexander Koller, and Giorgio Satta. "The Importance of Rule Restrictions in CCG". In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics.* Uppsala, Sweden: Association for Computational Linguistics, July 2010, pp. 534–543. URL: https://www.aclweb.org/anthology/P10-1055.

[MMS93] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Comput. Linguist.* 19.2 (June 1993), pp. 313–330. ISSN: 0891-2017. URL: http://dl.acm.org/citation.cfm?id=972470.972475.

[NG14] Rob Nederpelt and Professor Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. 1st. New York, NY, USA: Cambridge University Press, 2014. ISBN: 9781107036505.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[Ste00] Mark Steedman. *The Syntactic Process*. Cambridge, MA, USA: MIT Press, 2000. ISBN: 0-262-19420-1.

[VW90] K. Vijay-Shanker and David J. Weir. "Polynomial Time Parsing of Combinatory Categorial Grammars". In: *Proceedings of the 28th Annual Meeting on Association for Computational Linguistics*. ACL '90. Pittsburgh, Pennsylvania: Association for Computational Linguistics, 1990, pp. 1–8. DOI: 10.3115/981823.981824. URL: https://doi.org/10.3115/981823.981824.

[wik19a] OSM wiki. *Overpass language guide*. 2019. URL: https://web.archive.org/web/20190616125139/https://wiki.openstreetmap.org/wiki/Overpass_API/Language_Guide.

[wik19b] OSM wiki. *Overpass QL reference*. 2019. URL: http://web.archive.org/web/20190719122727/https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL.

[wik19c] OSM wiki. *Overpass turbo*. 2019. URL: https://web.archive.org/web/20190616124841/https://wiki.openstreetmap.org/wiki/Overpass_turbo.