Speagram Technical Documentation

Speagram Authors [1]

[1]All who contributed to Speagram until now: Łukasz Kaiser, Jacek Śliwerski, Andrzej Wasylkowski, Michał Ryszard Wójcik.

Contents

Introduction

Speagram is a tool for parsing and manipulating structured data. It gives an intuitive way to define expressive grammars for parsing tree-like structures (also known as terms or XML) and to operate on and transform such structures. It supports disambiguation of parses based on standard associativity and preference definitions as well as more advanced disambiguation. Operations on terms can be defined by rewrite rules which are easy to write for programmers (esp. functional programmers) and powerful enough to support any transformation of terms.

Speagram definitions of a grammar and term transformations can be used as an powerful parser for your grammar that will execute the defined operations after parsing. This is a convenient way to define languages for structured data and for getting the data parsed to xml documents. Since speagram grammars can be extended on the fly, anyone using your defined language will be able to extend it with new expressions and to define how these new expressions should be transformed to the declared xml datatype. Speagram also provides a meta-level interface where you can get more fine-grained control over the grammars and transformations and which can be used for further optimizations and analysis.

If you want to learn how to use speagram you should visit the page `www.speagram.org` and read the tutorial first. This document describes the ideas and algorithms used for speagram and is not meant as a tutorial. Most of these algorithms and ideas are well-known because speagram is based on term rewriting, polymorphic types and chart-based bottom-up parsing of context free grammars. Still we try to keep this documentation self-contained since it is the basis for implementation. Also we do not provide enough references - if you are interested in these techniques you should look for more information or email the authors of this document. We used a number of different sources when constructing speagram, but the first chapters of the book *Term Rewriting and All That* by F. Baader and T. Nipkow and the article *Functional Pearls: Functional Chart Parsing of Context Free Grammars* by Peter Ljunglöf were very helpful in clarifying the techniques.

Chapter 1

Types

When defining any language or grammar or even when just writing a program you almost always have to start by specifying what kind of objects you are going to operate on. You can do this either by defining classes in programming languages or by the use of non-terminal symbols in context-free grammars. In speagram types play this role, they are used to specify what group of things you are talking about. Still we are not happy with some flat enumeration of possible types, but we allow polymorphic types (sometimes called "generics"). In this way you can not only have simple types like *int* or *boolean* but also types with variables like $list(\alpha)$ meaning "list of something", so that you can then write both $list(int)$ and $list(boolean)$. Here we present the mathematical definitions for types.

1.1 Definition of Types

Let Θ be a countably infinite set of *type variables*. Let Γ be a finite set of *type symbols*. Let $h_1, h_2, h_3, \ldots$ be a sequence such that the sets Θ, Γ, $H = \{h_1, h_2, h_3, \ldots\}$ are pairwise disjoint. Let $\texttt{arity}\colon \Theta \cup \Gamma \cup H \to \{0, 1, 2, 3, \ldots\}$ be a function which assigns some arity to each type symbol and such that $\texttt{arity}(h_n) = n + 1$ and $\texttt{arity}(\alpha) = 0$ for each type variable $\alpha \in \Theta$.

The set of *types* is defined inductively as the smallest set $\mathcal{G}$ such that:

(1) $\Theta \subset \mathcal{G}$

(2) $\{T \in \Gamma\colon \texttt{arity}(T) = 0\} \subset \mathcal{G}$

(3) if $T \in \Gamma \cup H$, $\texttt{arity}(T) = n \geq 1$ and $R_1, \ldots, R_n \in \mathcal{G}$ then $T(R_1, \ldots, R_n) \in \mathcal{G}$.

In other words, the set of types is the set of all trees whose nodes contain elements of $\Theta \cup \Gamma \cup H$ and such that for each node the number of its children is equal to the arity of the element contained in the node. Leaves have elements with arity 0.

Let us adopt the notational convention that the type $h_n(R_1, R_2, \ldots, R_n, R)$ will be denoted as $R_1, R_2, \ldots, R_n \to R$. Such types will be called *functional types*. In other

words, a type is called a functional type if and only if its top symbol belongs to H. All other types will be called *simple types*.

Let us define the function $\texttt{typarity}\colon \mathcal{G} \to \{0, 1, 2, 3, \ldots\}$ in the following way:

(i) $\texttt{typarity}(T) = 0$ for each simple type $T \in \mathcal{G}$

(ii) $\texttt{typarity}(R_1, R_2, \ldots, R_n \to R) = n$ for functional types.

We are going to need the function `TypeVar` which returns the set of all type variables occurring in a given type:

(1) $\texttt{TypeVar}(x) = \{x\}$ for each type variable $x \in \Theta$

(2) $\texttt{TypeVar}Var(t) = \emptyset$ for each type $t \in \Gamma$ with $\texttt{arity}(t) = 0$

(3) $\texttt{TypeVar}(f(t_1, \ldots, t_n)) = \texttt{TypeVar}(t_1) \cup \ldots \cup \texttt{TypeVar}(t_n)$ otherwise.

1.2 Definition of Substitutions

For an arbitrary function $\sigma\colon \Theta \to \mathcal{G}$ put $Dom(\sigma) = \{\alpha \in \Theta\colon \sigma(\alpha) \neq \alpha\}$. Let $Subst = \{\sigma\colon \Theta \to \mathcal{G}\colon Dom(\sigma) \text{ is finite}\}$. An element of the set $Subst$ is called a *substitution*. The interpretation is that such a function substitutes types for type variables. Let $\sigma \in Subst$. Let us extend this function so that it can be applied to any type to obtain a new type with the type variables in the original type replaced according to the function σ. Let the function $\overline{\sigma}\colon \mathcal{G} \to \mathcal{G}$ be defined inductively in the following way:

(1) $\overline{\sigma}(\alpha) = \sigma(\alpha)$ for each $\alpha \in \mathcal{G} \cap \Theta$

(2) $\overline{\sigma}(T) = T$ for each $T \in \mathcal{G}$ with $\texttt{arity}(T) = 0$

(3) $\overline{\sigma}(T(R_1, \ldots, T_n)) = T(\overline{\sigma}(R_1), \ldots, \overline{\sigma}(R_n))$.

1.3 Composition of Substitutions

Let $\sigma, \delta\colon \Theta \to \mathcal{G}$ be two substitutions with their extensions $\overline{\sigma}, \overline{\delta}\colon \mathcal{G} \to \mathcal{G}$. Notice that

$$\{\alpha \in \Theta\colon \overline{\sigma} \circ \delta(\alpha) \neq \alpha\} \subset \{\alpha \in \Theta\colon \sigma(\alpha) \neq \alpha\} \cup \{\alpha \in \Theta\colon \delta(\alpha) \neq \alpha\},$$

where $f \circ g(x) = f(g(x))$. Hence, the set $\{\alpha \in \Theta\colon \overline{\sigma} \circ \delta(\alpha) \neq \alpha\}$ is finite and the function $\overline{\sigma} \circ \delta$ is a substitution. Now, we have the following formal relationship:

$$\overline{\sigma} \circ \overline{\delta} = \overline{\overline{\sigma} \circ \delta}.$$

Let the function $\overline{\sigma} \circ \delta$ be called the *composition* of the substitutions σ and δ.

Let $\sigma_1, \sigma_2\colon \Theta \to \mathcal{G}$ be substitutions. We say that σ_1 is *more general than* σ_2 if and only if there exists a substitution $\delta \in Subst$ such that $\overline{\delta} \circ \sigma_1 = \sigma_2$.

1.4 Unification of Types

Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a finite set of types. We say that these types can be *unified* if and only if there exists a substitution σ such that $\overline{\sigma}(R_1) = \overline{\sigma}(R_2) = \ldots = \overline{\sigma}(R_n)$. Such a substitution will be called a *unifier* for these types. Let $Uni(\mathcal{R})$ denote the set of all such unifiers. Formally,

$$Uni(\mathcal{R}) = \{\sigma \in Subst\colon \overline{\sigma}(\mathcal{R}) \text{ is a singleton}\},$$

where $\overline{\sigma}(\mathcal{R}) = \{\overline{\sigma}(R)\colon R \in \mathcal{R}\}$. Obviously, the types $R_1, \ldots, R_n$ can be unified if and only if $Uni(\{R_1, \ldots, R_n\}) \neq \emptyset$. Let

$$MGU(\mathcal{R}) = \{\sigma \in Uni(\mathcal{R})\colon (\forall \rho \in Uni(\mathcal{R}))\ \sigma \text{ is more general than } \rho\}.$$

Elements of $MGU(\mathcal{R})$ are called *most general unifiers* for the finite set of types $\mathcal{R}$.

Now, if $\mathcal{R} = \{R_1, \ldots, R_n\}$ and $\mathcal{S} = \{S_1, \ldots, S_m\}$ are two finite sets of types we may want to find a substitution that is a unifier for both $\mathcal{R}$ and $\mathcal{S}$ at the same time, in which case we would have $\overline{\sigma}(R_1) = \ldots = \overline{\sigma}(R_n)$ and $\overline{\sigma}(S_1) = \ldots = \overline{\sigma}(S_m)$. More generally, let $\{\mathcal{R}_1, \ldots, \mathcal{R}_N\}$ be a finite set of finite sets of types. Then we define

$$Uni2(\{\mathcal{R}_1, \ldots, \mathcal{R}_N\}) = Uni(\mathcal{R}_1) \cap \ldots \cap Uni(\mathcal{R}_N)\}.$$

And putting $\Psi = \{\mathcal{R}_1, \ldots, \mathcal{R}_N\}$, we define

$$MGU2(\Psi) = \{\sigma \in Uni2(\Psi)\colon (\forall \rho \in Uni(\Psi))\ \sigma \text{ is more general than } \rho\}.$$

We are going to present an algorithm for computing $MGU2(\{R_1, S_1\}, \ldots, \{R_n, S_n\})$.

We are going to prove two theorems about most general unifiers. First of all, if there exists a unifier then there exists a most general one. In other words, if $Uni(\mathcal{R}) \neq \emptyset$ then $MGU(\mathcal{R}) \neq \emptyset$. Furthermore, we are going to present an algorithm which computes a most general unifier for any finite set of types or declares that none exists. Secondly, we are going to prove that any two most general unifiers are identical up to variable renaming. More precisely, for every $\sigma_1, \sigma_2 \in MGU(\mathcal{R})$ there exists a substitution $\delta\colon \Theta \to \Theta$ such that $\sigma_1 = \overline{\delta} \circ \sigma_2$.

1.5 Unification Algorithm

Let $S = \{\{t_1, s_1\}, \ldots, \{t_n, s_n\}\} \subset \mathcal{P}(\mathcal{G})$. The algorithm below returns a substitution $subst \in MGU2(S)$ or fails when $Uni2(S)$ is empty. In this algorithm the substitution *subst* is represented as a finite list of items of the form $x \leftarrow t$ ($x \in \Theta$, $t \in \mathcal{G}$) where x is a type variable and t is a type substituted for that variable. The commands *apply* $x \leftarrow t$ *to* S and *apply* $x \leftarrow t$ *to subst* tacitly (and temporarily) introduce the substitution $\sigma \in Subst$ given by $\sigma(x) = t$ and $Dom(\sigma) = \{x\}$. The first command replaces S with $\{\{\overline{\sigma}(t_1), \overline{\sigma}(s_1)\}, \ldots, \{\overline{\sigma}(t_n), \overline{\sigma}(s_n)\}\}$ and the second command replaces *subst* with $\overline{\sigma} \circ subst$.

Algorithm 1 Algorithm for Computing MGU2

repeat
  for all $\{x, t\} \in S$ with $x \in \Theta$ **do**
    if $x = t$ **then**
      remove $\{x, t\}$ from S
    else
      if $x \in \texttt{TypeVar}(t)$ **then**
        FAIL
      else
        remove $\{x, t\}$ from S
        apply $x \leftarrow t$ to S
        apply $x \leftarrow t$ to *subst*
        append $x \leftarrow t$ to *subst*
      end if
    end if
  end for
  for all $\{f(t_1, \ldots, t_n), g(s_1, \ldots, s_m)\} \in S$ **do**
    if $f \neq g$ **then**
      FAIL
    else
      remove $\{f(t_1, \ldots, t_n), g(s_1, \ldots, s_n)\}$ from S
      append $\{t_1, s_1\}, \ldots, \{t_n, s_n\}$ to S
    end if
  end for
until $S = \emptyset$

1.6 Internal Type Names

We are going to introduce a mathematical function which takes a type (an element of $\mathcal{G}$) and returns a string (= a finite sequence of integers 32-127 interpreted as ASCII characters). This function will play no role in the theory of rewriting terms but will be essential in programming practice. The strings returned by this function will be interpreted as "names" of types — for purposes of developing and debugging Speagram code and for human-readable display.

We want this function to return such a name for a given type as to reflect its internal structure. It must represent the whole tree structure of the type and carry information whether a given node is a variable or not and whether a given node is a simple type or a functional type.

Let us refer to this name-assigning function as `TypeName`. We also need an auxiliary function `SymbolName` for names of type symbols. The strings returned by the function `SymbolName` must not contain any of the following characters: | @ | (|) | [|] | , | ‘ | ’ | |. If r is a type symbol with arity zero then let $\texttt{TypeName}(r) = \texttt{SymbolName}(r)$. Moreover, we want these two naming functions to be injective.

We demand that the name of any type variable begins in | @ | ? | | — followed by a nonempty string — and the rest of the name does not contain any of the following characters: | @ | (|) | [|] | , | ‘ | ’ | |.

By induction, we define the function `TypeName` for types with arity greater than zero.

If l is a type symbol with arity n then let $\texttt{TypeName}(l(t_1, \ldots, t_n)) = \texttt{SymbolName}(l) +$ | | (| $+ \texttt{TypeName}(t_1) +$ | , | | $+ \ldots +$ | , | | $+ \texttt{TypeName}(t_n) +$ |) |.

And functional types are dealt with in the following way:
$\texttt{TypeName}(h_n(t_1, \ldots, t_{n+1})) =$ | @ | F | | (| $+ \texttt{TypeName}(t_{n+1}) +$ | , | | $+$
$\texttt{TypeName}(t_1) +$ | , | | $+ \ldots +$ | , | | $+ \texttt{TypeName}(t_n) +$ |) |.

1.6.1 Internal Format Lexer

For certain very technical reasons we need to represent types and terms as strings within the workings of Speagram's innards. Below we present a lexer for dealing with such an internal format.

The lexer is defined by the following delimiters:

(	)	,	[	]	@F	@L	@V	@Y	@T	@:	@‘	@?

which means that to change a string of characters to a sequence of tokens we remove all white spaces and split on the above delimiters.

For example the name of the type $list(\alpha)$ consisting of a node with name *list* and arity 1 and a leaf being a type variable with name a is `list (@? a)`. When the lexer processes this name it returns the following sequence of tokens.

name	delimiter	delimiter	name	delimiter
`list`	`(`	`@?`	`a`	`)`

Chapter 2

Syntax Definitions

Let us fix a special element of the set of types: $term\_type \in \mathcal{G}$.

A *syntax definition* is a triple (a, b, c) such that

(1) $a \in \{$ type, constructor, function, variable $\}$,

(2) b is a finite sequence of elements which are either types or strings,

(3) if $a \neq$ type then $c \in \mathcal{G}$ (c is a type),

(4) if $a =$ type then b does not contain any types except $term\_type$
(the only type allowed in such a syntax definition is the special type $term\_type$),

(5) if $a =$ constructor then $c = head()$ or $c = head(\alpha_1, \ldots, \alpha_n)$
where $head$ is a type symbol and $\alpha_1, \ldots, \alpha_n$ are type variables.

Types are elements of the set $\mathcal{G}$ and strings are interpreted as ASCII characters. Notice that the set of types is disjoint from the set of strings.

Examples of syntax definitions

type	`boolean`	-
constructor	`true`	*boolean*
constructor	α `:` `:` *list*(α)	*list*(α)
function	α `equals` α	*boolean*
variable	`x`	*boolean*
variable	`multiple` `word` `variable` `name`	*boolean*
variable	`xx` *boolean*	*boolean*

2.1 Automatic Functional Syntax Definitions

To each syntax definition which declares a constructor, function or variable there corresponds a special functional syntax definition. Before we define this new concept, let us take a look at some examples of functional syntax definitions which correspond to some of the syntax definitions given earlier as examples.

constructor	{ } : : { }	$\alpha, list(\alpha) \rightarrow list(\alpha)$
function	{ } equals { }	$\alpha, \alpha \rightarrow boolean$
variable	{ x }	$boolean$
variable	xx { }	$boolean \rightarrow boolean$

Given a syntax definition (a, b, c) where $a \neq$ type, let us define the corresponding functional syntax definition as (A, B, C) such that

(1) $A = a$

(2) if there are no types in b then $C = c$

(3) if there is at least one type in b then C is the functional type $R_1, \ldots, R_n \rightarrow R$ where R_i's are the consecutive types appearing in b

(4) B contains only strings and the strings appearing in b are also preserved in B

(5) if b does not contain types then the first string in B is a new { and the last string is a new }

(6) else in place of types appearing in b,
we put the two strings { } as shown in the examples.

2.2 Generating Names from Syntax Definitions

We are going to introduce a mathematical function — called `GeneratedName` — which takes a syntax definition and returns a string (= a finite sequence of integers 32-127 interpreted as ASCII characters). This function will play no role in the theory of rewriting terms but will be essential in programming practice. The strings returned by this function will be interpreted as "names" of syntax definitions — for purposes of developing and debugging Speagram code and for human-readable display.

We want this function to return such a name for a given syntax definition as to reflect its internal structure. First of all, the strings present in a syntax definition will be preserved by the naming function with the following modifications to deal with special characters:

\	→	\\
_	→	_
(	→	\lb
)	→	\rb
[	→	\ls
]	→	\rs
,	→	\cm
‘	→	\qt
’	→	\ap
@	→	\at

The first character of each name of a syntax definition is determined in the following way: `T` for type, `C` for constructor, `F` for function and `V` for variable.

The rest of the name is formed by keeping the strings (with the modifications mentioned above), inserting \? for any type, and separating the list with underscore (_). In this way the generated name for the list constructor presented above is

```
C\?_:_:_\?
```

You should note that in this way names would not be unique and to prevent this we allow a suffix in the generated name that consists of an underscore (_) followed by an integer followed by a backslash. For example, if we have these two syntax definitions

constructor	α	:	:	$list(\alpha)$	$list(\alpha)$
constructor	int	:	:	$list(int)$	$list(int)$

then the name of the second one will have a suffix at the end:

```
C\?_:_:_\?_0\
```

Two names are *corresponding* when they differ only in the first letter and in the optional prefix. This correspondence will play an important role in defining preferences in the parser.

syntax definition						generated name
constructor	[	]			$list(\alpha)$	C\ls\rs
constructor	(	α_1	,	α_2	)	$pair(\alpha_1, \alpha_2)$	C\lb\?\cm\?\rb
function	int	a_maja	int	\psy	$boolean$	F\?_a_maja_\?_\\psy

It might not be obvious at first sight but you can check that by calculating whether an even or odd number of backslashes appears before special symbols and underscores it is possible to reconstruct — from a name generated in this way — the exact form of the syntax definition from which it was generated, and all the strings used in this syntax definition.

2.3 User Interface

Now we are going to define a special displaying function which takes two arguments and returns a string. The first argument is a finite sequence of strings some of which may be empty, say $A_1, \ldots, A_N$. The second argument is a finite sequence of nonempty strings, say $B_1, \ldots, B_M$.

The algorithm that computes the function can be roughly stated as follows. We go through the first sequence string by string from left to right (and simultaneously we look at consecutive strings from the second sequence and use them when necessary). If we find a nonempty string (in the first sequence) we display it and move on. If we find an empty string we display the next coming string from the second sequence or {} if we have already run out of the second sequence. When we have gone through the first sequence and there's still something left of the second sequence, we display the rest as a tuple: we enclose it in brackets and separate the strings with commas. Examples:

```
(ala,-,ba,-,cen,-,dak) (x)           --> ala x ba {} cen {} dak
(ala,-,ba,-,cen,-,dak) (x,yy)        --> ala x ba yy cen {} dak
(ala,-,ba,-,cen,-,dak) (x,yy,z1)     --> ala x ba yy cen z1 dak
(ala,-,ba,-,cen,-,dak) (x,yy,z1,u)   --> ala x ba yy cen z1 dak (u)
(ala,-,ba,-,cen,-,dak) (x,yy,z1,u,v) --> ala x ba yy cen z1 dak (u,v)

(-,:,:,-) (x,y) --> x : : y
(1,-,+,-) (2,3) --> 1 2 + 3
```

We want to make a modification to the displaying function given above. We want to avoid displaying spaces between two digits or between two non-alphanumeric characters. This is just a heuristic that makes things more readable in practice. The modification is very easy: before we put a space between two strings we first look at the last character of the first one and the first character of the second one. If both are digits, letters, both non-alphanumeric or these are an opening bracket (or [and and the second is alphanumeric, or the first is alphanumeric and the second a closing bracket, then we do not put a space between them.

```
(-,:,:,-) (x,y) --> x :: y
(1,-,+,-) (2,3) --> 12 + 3
```

It should be noted that given a syntax definition (or its generated name) we can create a corresponding finite sequence of strings to be used as the first argument for the displaying function described above.

2.4 Basic Syntax Definitions

In Speagram we use syntax definitions as the basic means of communication between the program and the user. You will normally enter objects in a natural syntax and Speagram will use the corresponding syntax definitions to parse them and create terms.

Terms are defined in the next section and they are the true objects manipulated by Speagram after parsing. But term symbols that are necessary to construct terms are just the names of the corresponding syntax definitions created by the `GeneratedName` function described above.

Before we go on to define terms we want to show you the most basic syntax definitions used in Speagram. The definitions presented here are the only ones built in the source code of Speagram, all other are user-defined. It might be unclear at that point what all these definitions mean, but it is good to look over them to get some intuition now. Moreover, we are going to refer to a number of special term and type symbols, e.g. *term_type* was already used before. The definitions presented below show the real representation that is used for such special elements.

```
Class ''bit''.
```

```
Element ''bit'' ''0'' as bit.
Element ''bit'' ''1'' as bit.

Class ''char''.
Element ''char'' ''code'' bit bit bit bit bit bit bit bit as char.

Class ''term'' ''type''.

Class term type ''list''.
Element ''['' '']'' as ?a list.
Element ?a '':'' '':'' ?a list as ?a list.

Class ''string''.
Element ''string'' ''from'' char list as string.

Class ''boolean''.
Element ''true'' as boolean.
Element ''false'' as boolean.

Class ''ternary'' ''truth'' ''value''.
Element ''true'' as ternary truth value.
Element ''unknown'' as ternary truth value.
Element ''false'' as ternary truth value.

// Term types (the special type of term types).
Element ''?'' string as term type.
Element ''type'' string '':'' term type list as term type.
Element ''funtype'' term type list ''-'' ''>'' term type as term type.

Class ''syntax'' ''element''.
Element ''''' ''''' string ''''' ''''' as syntax element.
Element term type as syntax element.
Class ''syntax'' ''element'' ''sequence''.
Element syntax element as syntax element sequence.
Element syntax element syntax element sequence as syntax element sequence.
Class ''syntax'' ''definition''.
Element ''class'' syntax element sequence as syntax definition.
Element ''element'' syntax element sequence ''as'' term type
    as syntax definition.
Element ''function'' syntax element sequence ''as'' term type
    as syntax definition.
Element ''variable'' syntax element sequence ''as'' term type
    as syntax definition.


Class ''term''.
```

```
Element ''var'' string '':'' term type ''('' term list '')'' as term.
Element ''term'' string ''('' term list '')'' as term.

Class ''constructor''.
Element ''constructor'' string ''from'' term type list ''to'' term type
    as constructor.

Class ''rewrite'' ''rule''.
Element ''rewrite'' term ''to'' term as rewrite rule.

Class ''input'' ''rewrite'' ''rule'' ''of'' term type.
Element ''let'' ?a ''be'' ?a as input rewrite rule of ?a.

Class ''priority'' ''input'' ''rewrite'' ''rule'' ''of'' term type.
Element ''let'' ''major'' ?a ''be'' ?a
    as priority input rewrite rule of ?a_1.

Class ''function'' ''definition''.
Element ''function'' string ''from'' term type list ''to'' term type
    as function definition.

Class ''class'' ''definition''.
Element ''class'' ''of'' term type as class definition.

Class term type ''exception''.
Element ''!'' ''!'' ?a ''!'' ''!'' as ?other_than_a! exception.
Element ''+'' ''+'' ?a ''+'' ''+'' as ?a exception.

// If-then-else.
Function ''if'' boolean ''then'' ?a ''else'' ?a as ?a.

// Bracketing = identity.
Function ''('' ?b '')'' as ?b.

// Verbatim function explained later.
Function ''<'' ''|'' ?b ''|'' ''>'' as ?b.

// Preference function used for disambiguating parses.
Function term ''parsed'' ''preferred'' ''to'' term as ternary truth value.

// Preprocessing function to customize the parser.
Function ''#'' ''#'' ''#'' ?p as ?q.


// Meta-level functions allow changing the system dynamically.
Function ''code'' ?a ''as'' ''term'' as term.
```

```
Function ''decode'' term ''with'' ''type'' ''as'' ?a as ?a.

Function ''get'' ''class'' ''definitions'' as class definition list.
Function ''get'' ''function'' ''definitions'' as function definition list.
Function ''get'' ''constructors'' as constructor list.


// Special class used for loading files.
Class ''outside'' ''paths''.
Element ''library'' '':'' ''/'' string as outside paths.
Element ''file'' '':'' ''/'' string as outside paths.
Class ''load'' ''command''.
Element ''load'' ''state'' outside paths as load command.

// Closing context removes visible variables from scope.
Class ''system'' ''commands'' ''of'' term type.
Element ''close'' ''context'' as system commands of ?a.
```

Chapter 3

Terms with Types

In the previous section we defined classes of objects so now we can proceed to defining the objects i.e. terms. You can imagine a term as any object that finally belongs to some type. For example 1 is a term of type *int* and similarly $1+2$ sometimes written as $+(1,2)$ is also of type *int*. It might also happen that you will have variables inside terms, like in $1+x$. Symbols and variables have associated types, but it might be unclear what is actually the type of the whole terms. Moreover, the types for variables might be unspecified and then these have to be reconstructed. We present here the necessary definitions and algorithms to handle terms with types.

3.1 Definition of Terms

In order to define terms we need the set $\mathcal{G}$ of *types* and the function $\texttt{typarity}\colon \mathcal{G} \to \{0,1,2,3,\ldots\}$ defined earlier.

Let V be a countably infinite set of *term variables*. Let Σ be a finite set of *term symbols*. We demand that the sets $\mathcal{G}$, V, Σ are pairwise disjoint. Let $\texttt{type}\colon \Sigma \to \mathcal{G}$ be a function which assigns a fixed type to each term symbol. We do not assign types to term variables. We assign arity to each term symbol $s \in \Sigma$ by $\texttt{arity}(s) = \texttt{typarity}(\texttt{type}(s))$. We do not assign arity to term variables.

Let t be a tree whose nodes contain elements of $V \cup \Sigma$ and let $vtype\colon V \to \mathcal{G}$ be a function which assigns a type to each term variable. Then for each node in this tree we can define its arity in the following way: if the node contains a term symbol $s \in \Sigma$ then the arity of the node is equal to $\texttt{arity}(s)$ and if it contains a term variable $x \in V$ then the arity of the node is equal to $\texttt{typarity}(vtype(x))$.

Now, the ordered pair $(t, vtype)$ is called a *term* if and only if for each node that is not a leaf the number of its children is equal to its arity. Notice that we place no demands on the arity of leaves.

We can think that each node of a term has an inherent type: either through the global function $\texttt{type}\colon \Sigma \to \mathcal{G}$ (when it contains a term symbol) or through the local function $vtype\colon V \to \mathcal{G}$ (when it contains a term variable).

If s is a subtree of t then we say that $(s, vtype)$ is a sub-term of $(t, vtype)$. Whenever the context is clear we will simply write s or t to refer to the terms $(s, vtype)$ and $(t, vtype)$ respectively. We will also write that s is a sub-term of t.

We are going to need the function `TermVar` which returns the set of all term variables occurring in a given term:

(1) $\texttt{TermVar}(x) = \{x\}$ for each term variable $x \in V$

(2) $\texttt{TermVar}(t) = \emptyset$ for each term symbol $t \in \Sigma$ with $\texttt{arity}(t) = 0$

(3) $\texttt{TermVar}(f(t_1, \ldots, t_n)) = \texttt{TermVar}(t_1) \cup \ldots \cup \texttt{TermVar}(t_n)$ if f is a term symbol

(4) $\texttt{TermVar}(x(t_1, \ldots, t_n)) = \{x\} \cup \texttt{TermVar}(t_1) \cup \ldots \cup \texttt{TermVar}(t_n)$ if x is a term variable.

3.2 Definition of Well-Typed Terms

A *typing* of a term $(t, vtype)$ is any function which assigns a type to each of its sub-terms. The types which each sub-term receives from such a function should not be confused with the types of the nodes inherent in that term.

Let Ψ be a typing of a term $(t, vtype)$. We say that Ψ is *coherent* if and only if for each sub-term u the following condition holds:

(i) if u is a leaf containing a term variable $x \in V$ then $\Psi(u) = vtype(x)$

(ii) if $u = x(t_1, \ldots, t_n)$ where $n \geq 1$, $x \in V$ and $vtype(x) = \alpha_1, \ldots, \alpha_n \rightarrow \beta$ then $\Psi(u) = \beta$ and $\Psi(t_i) = \alpha_i$ for each $i = 1, \ldots, n$

(iii) if u is a leaf containing a term symbol $h \in \Sigma$ then there exists a substitution $\sigma \in Subst$ such that $\Psi(u) = \overline{\sigma}(\texttt{type}(h))$

(iv) if $u = h(t_1, \ldots, t_n)$ where $n \geq 1$, $h \in \Sigma$ and $\texttt{type}(h) = \alpha_1, \ldots, \alpha_n \rightarrow \beta$ then there exists a substitution $\sigma \in Subst$ such that $\Psi(u) = \overline{\sigma}(\beta)$ and $\Psi(t_i) = \overline{\sigma}(\alpha_i)$ for each $i = 1, \ldots, n$.

We say that the term $(t, vtype)$ can be *well-typed* if and only if there exists a coherent typing for this term. We say that *the types of the term variables of t can be reconstructed* if and only if there exists a substitution $\rho \in Subst$ such that the term $(t, \overline{\rho} \circ vtype)$ can be well-typed.

If $E \subset Subst$ then let $MostGeneral(E) = \{\rho \in E \colon (\forall \sigma \in E)\, \rho \text{ is more general than } \sigma\}$. If Ψ_1 and Ψ_2 are two typings of the term $(t, vtype)$ then we say that Ψ_1 *is more general than* Ψ_2 if and only if there exists a substitution $\sigma \in Subst$ such that $\overline{\sigma} \circ \Psi_1 = \Psi_2$.

In the next section we are going to present an algorithm which takes an arbitrary term $(t, vtype)$ and computes a most general substitution ρ such that $(t, \overline{\rho} \circ vtype)$ can be well-typed and a most general coherent typing of the term $(t, \overline{\rho} \circ vtype)$, or fails if the term cannot be well-typed.

Recall that for a term which can be well-typed there exists a most general coherent typing. This typing — being a function assigning a type to each subterm — can be used to assign a type to such a term by simply taking the type of the whole term. Since a most general coherent typing is not unique (although it is unique up to type variable renaming), this type assignment is not unique, either. However, let us select one of the possible outcomes of the type assigning procedure to obtain a function `TermType` assigning types to terms that can be well-typed. From now on, when we refer to the type of a term we are tacitly referring to the function `TermType`. Furthermore, terms that can be well-typed will be called *typed terms*.

3.3 Type Reconstruction Algorithm

3.3.1 Type Variable Renaming

We say that a substitution $\sigma \in \mathit{Subst}$ is a *type variable renaming* if and only if $\sigma(\alpha) \in \Theta$ for every $\alpha \in \Theta$ and $\sigma\colon \Theta \to \Theta$ is a bijection. If σ is a type variable renaming then let $\mathtt{Range}(\sigma) = \{\sigma(\alpha)\colon \alpha \in \mathtt{Dom}(\sigma)\}$.

Since the set of type variables Θ is infinite, we have the following theorem: for any finite $A \subset \Theta$ and any finite $R \subset \Theta$ there exists a type variable renaming σ such that

(1) $\mathtt{Dom}(\sigma) = R$

(2) $\mathtt{Range}(\sigma) \cap A = \emptyset$.

In our algorithm we will have the following situation: a finite set of type variables $A_0 \subset \Theta$ and a sequence of finite sets of type variables $R_1, R_2, \ldots, R_n \subset \Theta$. These sets will not necessarily be pairwise disjoint. We will be interested in "renaming" the sets in our sequence (thus obtaining a new sequence $R'_1, \ldots, R'_n$) so that the sets $A_0, R'_1, R'_2 \ldots, R'_n$ are pairwise disjoint. In the first step we will make use of the type variable renaming σ_1 such that $\mathtt{Dom}(\sigma_1) = R_1$ and $\mathtt{Range}(\sigma_1) \cap A_0 = \emptyset$ and in the k-th step we will use the type variable renaming σ_k such that $\mathtt{Dom}(\sigma_k) = R_k$ and $\mathtt{Range}(\sigma_k) \cap (A_0 \cup R_1 \cup \ldots \cup R_{k-1}) = \emptyset$. Then $R'_k = \{\sigma_k(\alpha)\colon \alpha \in R_k\}$.

3.3.2 The Algorithm

We will present the algorithm in three steps. Additionally, we will illustrate its workings by following what it does with two example terms. Naturally, the examples are extra — the algorithm itself is presented with sufficient rigor.

We will use the following symbols to construct the two examples:

type symbol	arity
int	0
bool	0
list	1
pair	2

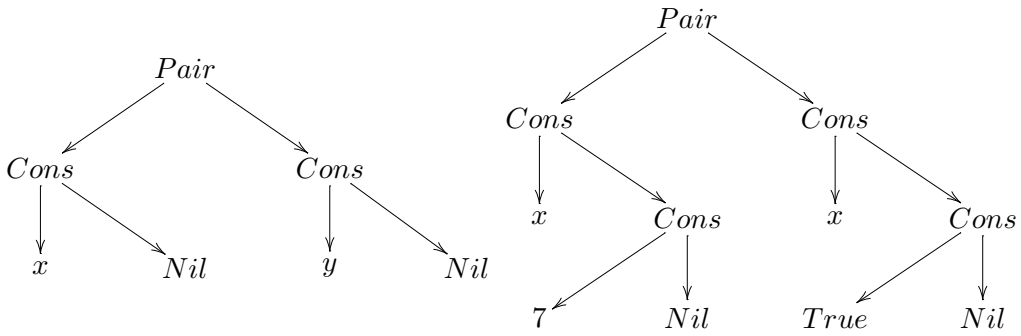
term symbol	type	typarity
7	int	0
True	bool	0
Pair	a,b → pair(a,b)	2
Cons	a,list(a) → list(a)	2
Nil	list(a)	0

And the following variables will be used:

type variables	term variables
a,b,c,d,e,f,g,h,i,j	x,y

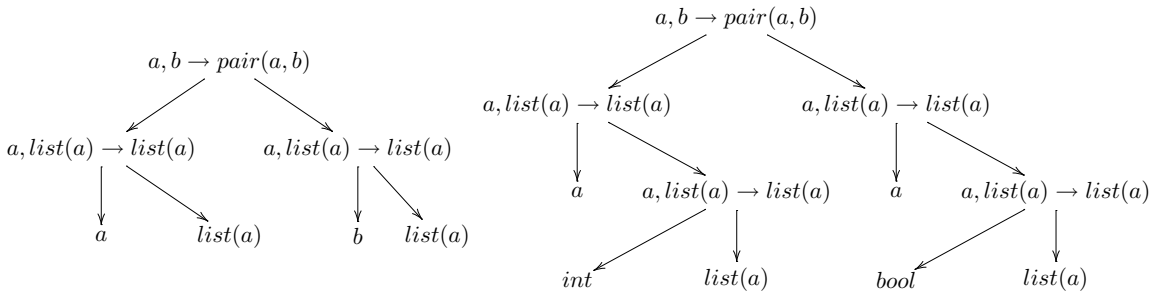
The two examples are:
(1) Pair(Cons(x,Nil),Cons(y,Nil)),
(2) Pair(Cons(x,Cons(7,Nil)),Cons(x,Cons(True,Nil))),
where *vtype* in both cases is such that $vtype(x) = a$ and $vtype(y) = b$.

These terms can be depicted in the following way:



As its input, the algorithm takes an arbitrary term $(t, vtype)$ where t is a tree according to the definition of a term. We are going to need an isomorphic tree ψ whose nodes contain the types of the elements in the corresponding nodes of the tree t. The types of elements of t are determined by the global function `type` for term symbols and by the local function *vtype* for term variables occurring in t.

The resulting ψ-trees for our example terms can be depicted as follows.

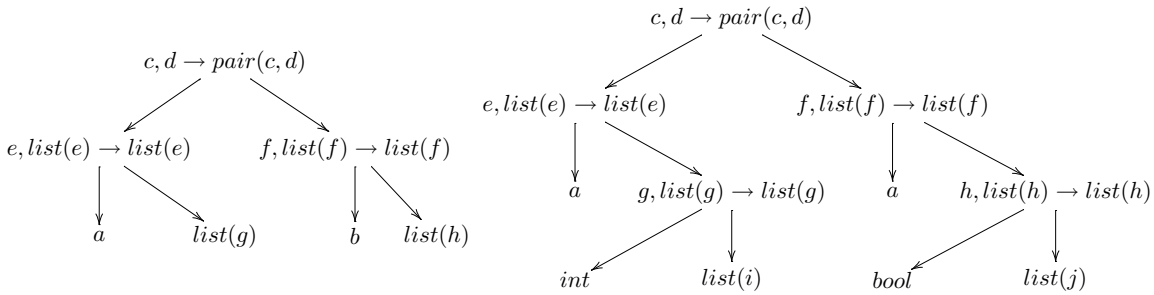


Let `RECOVARS` = `TypeVar`$(vtype($`TermVar`$(t)))$ $\subset \Theta$. Notice that `RECOVARS` is a finite set of type variables. When the algorithm returns the substitution ρ which reconstructs the types of the term variables in t we will have $Dom(\rho) \subset$ `RECOVARS`.

In the first example, `RECOVARS` $= \{a, b\}$; and in the second example, `RECOVARS` $= \{a\}$.

STEP ONE:
Let n be the number of nodes in t containing a term symbol. Let $R_1, \ldots, R_n$ denote the sets of type variables occurring in those nodes of ψ whose corresponding nodes in t contain a term symbol. Let $A_0 =$ `RECOVARS`. Perform the variable renaming procedure described above to obtain a new tree ψ_1 of types isomorphic to ψ such that the sets of type variables occurring in those renamed nodes are pairwise disjoint and each of them is disjoint from `RECOVARS`.

The result of the variable renaming can be depicted as follows.



STEP TWO:
Let W be the set of thus renamed nodes of ψ_1 minus the set of leaves.

Algorithm 2 Type Reconstruction Algorithm: Step Two

```
let BIGSACK = ∅
for all α_1, ..., α_n → β ∈ W do
  for all i ∈ {1, ..., n} do
    let ... → β_i denote the type of the i-th child
    append {α_i, β_i} to BIGSACK
  end for
end for
find σ ∈ MGU2(BIGSACK) or fail
```

In the first example, BIGSACK =

$$\{\{c, list(e)\}, \{d, list(f)\}, \{e, a\}, \{list(e), list(g)\}, \{f, b\}, \{list(f), list(h)\}\}$$

and $\sigma \in$ MGU2(BIGSACK) can be represented as

$$\{c \leftarrow list(a), d \leftarrow list(b), e \leftarrow a, f \leftarrow b, g \leftarrow a, h \leftarrow b\}.$$

In the second example, BIGSACK =

$$\{\{c, list(e)\}, \{d, list(f)\}, \{e, a\}, \{list(e), list(g)\},$$

$$\{f, a\}, \{list(f), list(h)\}, \{g, int\}, \{list(g), list(i)\},$$

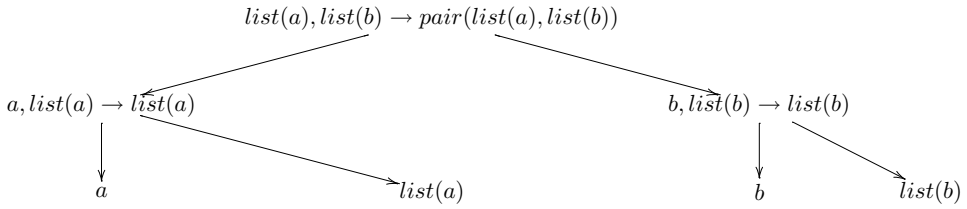
$$\{h, bool\}, \{list(h), list(j)\}\}$$

and MGU2(BIGSACK) is empty because ultimately we run against $int \neq bool$.

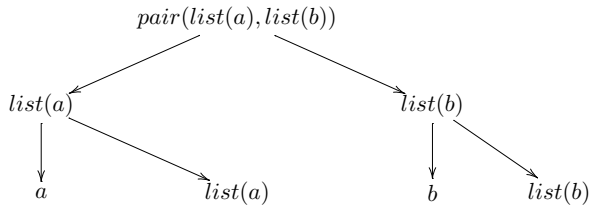
STEP THREE:
Let $\rho(\alpha) = \sigma(\alpha)$ for each $\alpha \in$ `RECOVARS` and $\rho(\alpha) = \alpha$ otherwise. This ρ is the substitution returned by the algorithm. We say that it reconstructs the types of the term variables: we now look at the term $(t, \overline{\rho} \circ vtype)$ which differs from the original term $(t, vtype)$ in that the types of the term variables are now such that there exists a coherent typing for the new term.

Let the coherent typing (returned by the algorithm) be the result of collapsing the functional types in $\overline{\sigma} \circ \psi_1$ except in the leaves, where $\texttt{collapse}(\alpha_1, \ldots, \alpha_n \to \beta) = \beta$.

The new tree of types $\overline{\sigma} \circ \psi_1$ can be depicted as



and after the collapsing procedure we get the representation of the coherent typing which is ultimately returned by the algorithm:



3.4 Internal Term Display Format

Before we start with the display format we have to note that things can get coded as term. In fact each basic syntax definition presented in the previous chapter gives rise to two function that code and decode objects in the source code or in speagram as appropriate terms and decode them back from terms. These functions are a part of speagram and are used throughout this document, for example these are used when we need to distinguish if a term represents a list or a string or a syntax definition etc.

In a similar way to types we represent terms in an internal format in the following way.

(1) if t represents a string s then we write `@'s@'`,

(2) if t represents a list $t_1, \ldots, t_n$ then we write `@L[`$t_1, \ldots, t_n$`]`,

(3) if t represents a type T then we write `@Y` T using internal representation of types for T,

(4) if t encodes another term t' then we write `@T` t' using this procedure for t' again,

(5) to display $x(t_1, \ldots, t_n)$ for a variable symbol x with $\mathrm{vtype}(x) = T$ we print `@V [`x `@:` T `] (`$t_1, \ldots, t_n$`)`,

(6) to display $f(t_1, \ldots, t_n)$ we use the standard notation `f (`$t_1, \ldots, t_n$`)`.

The special cases for lists, strings, types and terms are used because it makes the internal format more readable for larger terms.

Chapter 4

Ground Rewriting

4.1 Rewrite Rules

Since we want to transform terms, we need to define programs — transformations acting on terms. Let the set of term symbols be divided into two disjoint subsets: the set of *constructors* and the set of *function names*. Normally, functions will be transformed and constructors will form our data-types.

To define a function in a program that we want to execute we are going to need the concept of a *rewrite rule* — a pair of terms l and r, the left and right side of the rule, denoted by $l \to r$.

4.1.1 Definition of Rewrite Rule

An ordered pair of two typed terms $(l, vtype)$ and $(r, vtype')$ is called a *rewrite rule* if and only if the following conditions hold:

(1) $\texttt{TermType}(l) = \texttt{TermType}(r)$,

(2) $\texttt{TermVar}(r) \subset \texttt{TermVar}(l)$,

(3) $(\forall x \in \texttt{TermVar}(r))\ vtype(x) = vtype'(x)$,

(4) all term variables occurring in l are located in the leaves,

(5) the term symbol at the top position in l is a function name,

(6) there are no function names in l except at the top position.

4.1.2 Substitution for Term Variables

Before we define substitutions for term variables we need to introduce the concept of a term tree.

A *term tree* is a tree whose nodes contain term symbols or term variables and which satisfies the following condition: for each node that is not a leaf if it contains a term symbol then the number of this node's children is equal to the arity of that term symbol. Notice that we place no demands on the arity of leaves, and no demands on nodes containing term variables. Also notice that depending on the choice of a function $vtype\colon V \to \mathcal{G}$ the pair (this tree, $vtype$) is a term or not.

A function s — which assigns a term tree to each term variable — such that the set $Dom(s) = \{x \in V\colon s(x) \neq x\}$ is finite will be called a *substitution for term variables*. If s is a substitution for term variables, let $\overline{s}$ be a partial function on the set of term trees into the set of term trees defined in the following way:

(i) $\overline{s}(x) = s(x)$ if x is a term variable,

(ii) $\overline{s}(f(t_1, \ldots, t_n)) = f(\overline{s}(t_1), \ldots, \overline{s}(t_n))$ if f is a term symbol or f is a term variable such that $f \notin Dom(s)$,

(iii) $\overline{s}(x(t_1, \ldots, t_n)) = f(\overline{s}(t_1), \ldots, \overline{s}(t_n))$ if $s(x) = f$ where x is a term variable and f is a term variable or a term symbol with arity n.

4.2 Application of Rewrite Rules

The rewrite rule $(l, vtype_1) \to (r, vtype_2)$ can be *applied* to the typed term $(t, vtype)$ if and only if there exists a substitution for term variables s such that $\overline{s}(l) = t$, in which case the result of this application is defined to be $(\overline{s}(r), vtype)$, which is a typed term with the same type as the original term $(t, vtype)$.

Notice that for any two s_1, s_2 such that $\overline{s_1}(l) = \overline{s_2}(l) = t$ it holds that $\overline{s_1}(r) = \overline{s_2}(r)$. Proof. Take any $x \in \texttt{TermVar}(r)$. Since $\texttt{TermVar}(r) \subset \texttt{TermVar}(l)$, $x \in \texttt{TermVar}(l)$, which allows us to conclude that since $\overline{s_1}(l) = \overline{s_2}(l)$, it holds that $s_1(x) = s_2(x)$. We showed that for every $x \in \texttt{TermVar}(r)$ it holds that $s_1(x) = s_2(x)$ — hence $\overline{s_1}(r) = \overline{s_2}(r)$.

It still remains to demonstrate why the result of the application of a rewrite rule to a typed term is a typed term with the same type.

4.2.1 Rewriting — choosing an appropriate rewrite rule

Let f be a function name and let $\mathcal{R}$ be a finite set of all those rewrite rules in the system that have f at the top position in the left-hand term. Let us consider a typed term with f at the top position. In this section we will discuss the procedure of deciding which rewrite rule from the set $\mathcal{R}$ should be applied to rewrite this term.

First, we will discuss the simplest case: when the term contains no term variables and no function names except f at the top position. Then we will move on to the more complex situation when there are term variables present, and finally we will cover the case of function names appearing in the term to be rewritten.

No function names and no term variables

Suppose for the time being that our term contains no term variables and no function names except f at the top position. Now we want to describe the procedure of deciding whether this term should be rewritten (whether there is a rewrite rule that should be applied to this term).

First of all, we impose a linear order on the set $\mathcal{R}$ so that the first rewrite rule is more important than the second and so forth. Now, we check whether the first rule can be applied to this term according to the definition given above. If so, then we apply this rule and rewrite the term. If not, then we move on to the next rule according to the linear order. When we have tried all the rewrite rules and none coould be applied, the term cannot be rewritten.

The algorithm for deciding whether a given rewrite rule can be applied to a given term with no function names and no variables is quite straightforward. Visually, let's have the left-hand side term of the rewrite rule on the left and the term to be rewritten on the right. Now, we traverse the two terms and compare them position by position. If we find a syntactic difference (= different constructors at corresponding positions) we stop and report that the rewrite rule cannot be applied because, naturally, in such a situation there does not exist a substitution required by the definition. And if we find a variable on the left we gather the information that the subtree on the right at the corresponding position should be substituted for this variable. Since this variable can appear multiple times, it can happen that two or more term trees would have to be substituted for this variable. In this case we report that the rule should be rejected. Otherwise, the rule should be applied and we have constructed the necessary substitution in the meantime. (See Algorithm 3.)

4.3 Normalisation

We should describe ground normalisation here.

Algorithm 3 Rewriting a term with no function names and no variables

STEP ONE
Let $SUBST = \emptyset$ and let $SubstVar = \emptyset$.
Let W be the set of positions that appear in both term trees l and t.
for all $p \in W$ **do**
  Let a be the subtree of l at position p.
  Let b be the subtree of t at position p.
  if a is a variable **then**
    $SUBST := SUBST \cup \{a \leftarrow b\}$
    $SubstVar := SubstVar \cup \{a\}$
  else
    if the top symbols of a and b are different **then**
      report REJECT and stop
    end if
  end if
end for

STEP TWO
for all $x \in SubstVar$ **do**
  Let $A = \{\omega \colon x \leftarrow \omega \in SUBST\}$.
  if A has more than one element **then**
    report REJECT and stop
  end if
end for
report APPLY and return $SUBST$

Chapter 5

Relations and Logic

In the previous chapters we defined the basis for Speagram — well-typed terms — and we have shown how to rewrite them and how to communicate them to Speagram using the parser. Now we want to make our comprehension of well typed terms complete by defining the whole logic on them and investigating reasoning.

5.1 First-order Logic

We are going to use the standard first-order logic on well-typed terms, here we just repeat the basic definitions.

!!! TODO; FIXME; We have functional types !!! !!! These are interpreted by rewrite rules lists !!!

First-order logic gives a universal way to express conditions and properties of elements of mathematical models, which in our case are well typed terms reduced through normalisation. To achieve this we use *formulas* that are defined as follows

$$\varphi \;:=\; (t_1 = t_2) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists x\ \varphi \mid \forall x\ \varphi,$$

where t_1 and t_2 are any well-typed terms.

When using formulas in practice we adhere to a few convenient conventions. First of all, we use $\varphi_1 \to \varphi_2$ and $\varphi_1 \leftrightarrow \varphi_2$ as shorthands for formulas with the meaning

$$\varphi_1 \to \varphi_2 := (\neg\varphi_1) \vee \varphi_2, \quad \varphi_1 \leftrightarrow \varphi_2 := (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1).$$

Additionally, you will see that $\wedge$ and $\vee$ are associative and therefore we do not put brackets when they appear multiple times. To make notation even simpler we assume that quantifiers $(\exists, \forall)$ bind stronger than negation $(\neg)$, which binds stronger than conjunction $(\wedge)$ and disjunction $(\vee)$, which in turn bind stronger than implication $(\to)$ and equivalence $(\leftrightarrow)$. For example $\exists x\ \psi_1 \wedge \neg\psi_2 \to \psi_3 = ((\exists x\ \psi_1) \wedge (\neg\psi_2)) \to \psi_3$. You should note that with conjunction and disjunction we must use brackets, so for example $\psi_1 \wedge \psi_2 \vee \psi_3$ is ambiguous, not well defined.

To the syntactic definition of formulas there is an associated notion of *free variables* of a formula φ, denoted $FV(\varphi)$ and defined inductively:

$$FV(\varphi) = \begin{cases} \mathrm{TermVar}(t_1) \cup \mathrm{TermVar}(t_2) & \text{when} \quad \varphi = (t_1 = t_2), \\ FV(\psi) & \text{when} \quad \varphi = \neg\psi, \\ FV(\varphi_1) \cup FV(\varphi_2) & \text{when} \quad \varphi = \varphi_1 \wedge \varphi_2, \\ FV(\varphi_1) \cup FV(\varphi_2) & \text{when} \quad \varphi = \varphi_1 \vee \varphi_2, \\ FV(\psi) \setminus \{x\} & \text{when} \quad \varphi = \exists x\ \psi, \\ FV(\psi) \setminus \{x\} & \text{when} \quad \varphi = \forall x\ \psi. \end{cases}$$

For a formula φ with $FV(\varphi) = \{x_1, \ldots, x_n\}$ and a tuple of terms $\overline{t} = \{t_1, \ldots, t_n\}$ we define the substituted formula $\sigma_{\overline{t}}$ as the formula φ with every variable x_i substituted by t_i. It should be clear what this means as substitutions on terms were defined before, we assume additionally that $\texttt{TermType}(t_i) = \mathrm{type}(x_i)$, and you need to remember that each variable has an associated type.

To understand what the formulas *mean* we have to define the *semantics* of formulas. To achieve this we associate with every formula φ a set of tuples of well-typed terms denoted $[\![\varphi]\!]$. We say that for these tuples the formula *is true* or *holds*. With every formula φ we associate tuples of length $FV(\varphi)$ and we first need to say that a formula $t_1 = t_2$ *holds trivially* when both t_1 and t_2 can be normalised in a finite number of steps and the results are equal. Let us now define the semantics inductively:

$$\overline{t} \in [\![\varphi]\!] \Leftrightarrow \begin{cases} \sigma_{\overline{t}}(\varphi) \text{ holds trivially} & \text{when} \quad \varphi = (t_1 = t_2), \\ \overline{t} \notin [\![\psi]\!] & \text{when} \quad \varphi = \neg\psi, \\ \overline{t} \in [\![\varphi_1]\!] \text{ and } \overline{t} \in [\![\varphi_2]\!] & \text{when} \quad \varphi = \varphi_1 \wedge \varphi_2, \\ \overline{t} \in [\![\varphi_1]\!] \text{ or } \overline{t} \in [\![\varphi_2]\!] & \text{when} \quad \varphi = \varphi_1 \vee \varphi_2, \\ \text{there is a well-typed term } s \text{ so that } s, \overline{t} \in [\![\varphi]\!] & \text{when} \quad \varphi = \exists x\ \psi, \\ \text{for all well-typed terms } s \text{ it holds } s, \overline{t} \in [\![\varphi]\!] & \text{when} \quad \varphi = \forall x\ \psi. \end{cases}$$

Very often instead of writing $\overline{t} \in [\![\varphi]\!]$ we are going to use the *model* ($\models$) relation,

$$\overline{t} \models \varphi \iff \overline{t} \in [\![\varphi]\!].$$

The way we use this relation is the same as it is usually used in mathematics, but normally you have to specify what the universe is, and we implicitly assume that the universe are all well-typed terms.

When interested in free variables of a formula φ we are going to use the notation $\varphi(\overline{x})$ which indicates that $FV(\varphi) \subseteq \overline{x}$. For two formulas φ and ψ with $|FV(\varphi)| = |FV(\psi)|$ we are going to use the usual extension of the model relation and say that

$$\varphi \models \psi \iff \text{for all } \overline{t} \text{ when } \overline{t} \models \varphi \text{ then } \overline{t} \models \psi.$$

In such case, when $\varphi \models \psi$, we say that φ *implies* or *semantically implies* ψ. We say that φ and ψ are *equivalent* when $\varphi \models \psi$ and $\psi \models \varphi$.

5.2 Defining Relations

Equipped with the notions from logic let us now define a new type of objects in Speagram — *relations*. We define a relation $R(\overline{x})$ by giving a formula φ of first-order

logic with $FV(\varphi) \subseteq \overline{x}$. When talking about a relation R we denote the associated defining formula by φ_R and for a tuple of terms $\overline{t}$ we say that $R(\overline{t})$ *holds* when $\overline{t} \models \varphi_R$.

Chapter 6

Term Simplification

We covered ground rewriting in a chapter before. Now we proceed to simlpifying terms where variables might appear, i.e. this what Speagram in fact does.

6.1 Rewriting — choosing an appropriate rewrite rule

No function names but term variables allowed

Now, let us consider a typed term with f at the top position without excluding the possibility that it contains term variables, but let it still contain no function names other than f at the top position. We will demonstrate the problem of deciding whether this term should be rewritten with a lucid example.

Suppose we have these two rewrite rules:

(1) $and(true, true) \rightarrow true$,

(2) $and(x, y) \rightarrow false$ with $vtype(x) = vtype(y) = bool$.

And let our example term be $and(z, true)$ with $vtype(z) = bool$.

You should notice that the first rule cannot be applied according to the definition given above because there is no substitution for term variables s such that $\overline{s}(and(true, true)) = and(z, true)$. If we move on to the next rule, we notice that it can be applied through the substitution $\{x \leftarrow z; y \leftarrow true\}$ to yield the result $false$. But, naturally, we don't want our system to rewrite the term $and(z, true)$ into the term $false$.

The following definition will be used to decide such cases. Let $l \rightarrow r$ be a rewrite rule and let t be a typed term such that $\texttt{TermVar}(l) \cap \texttt{TermVar}(t) = \emptyset$. We say that the rewrite rule $l \rightarrow r$ *clogs* on the typed term t if and only if there is no substitution for term variables s such that $\overline{s}(l) = t$ but there is a substitution s such that $\overline{s}(l) = \overline{s}(t)$.

It is important that the term variables of the left-hand term of the rewrite rule should be disjoint from the term variables of the term to be rewritten. Consider this example:

if $f(C(x)) \to h(x)$ is a rewrite rule and $f(x)$ is a term to be rewritten, then in fact we have a clog (= we could potentially apply this rule but we're not sure — after all, the x in $f(x)$ could be of the form $C(z)$ for some z) but according to the definition there is no clog.

Now, the procedure for deciding whether we can rewrite a typed term with f at the top position and no function names anywhere else looks like this:

Try consecutive rewrite rules from the set $\mathcal{R}$ according to the linear order of importance. If you encounter a clog, do not rewrite the term and do not try other rules. Just stop. If you can apply the rule, then rewrite the term and do not try other rules. Otherwise, go on to the next rule and repeat.

The definition of clog relies on two notions: matching and unification. We have a clog when there is no matching but unification is possible. Unfortunately, if we wanted to implement clog detection according to this definition we would have a rather slow-working system for rewriting terms. So our rewriting algorithm is a bit different from the one which would fully respect the mathematical definition of clog. First of all, in the case of a term to be rewritten which contains no term variables it works ideally. However, in the case of a term with term variables it might happen that our algorithm will report a clog and thus fail to try other rewrite rules, when in fact there is no clog and other rules could be tried. When this is not the case, the algorithm rewrites properly — there are no rewriting errors — the only flaw is that sometimes it might happen that it stalls and leaves the term untouched when in fact it could be rewritten. But it does not stall in many practical cases where each variable occurs only once and it is much faster than the complete algorithm, therefore we consider it a fair tradeoff.

6.1.1 Description of the Rewriting Algorithm

The algorithm for deciding whether a given rewrite rule can be applied to a given term with no function names but with variables is a straightforward modification of the algorithm for the case with no variables. As before, let's have the left-hand side term of the rewrite rule on the left and the term to be rewritten on the right. Now, we traverse the two terms and compare them position by position.

If we find a syntactic difference (= different constructors at corresponding positions) we stop the algorithm and report that the rewrite rule cannot be applied and should be rejected because there is no possibility of clog since there does not exist a matching substitution required by the definition of clog.

If we find a variable on the right and a constructor on the left, we report a (potential) clog because the variable on the right could potentially assume such a value as to correspond to the tree on the left. (This is the situation when we might be wrongly reporting a clog.) Once a clog has been reported there is no chance of accepting the rewrite rule by the algorithm. However, we must continue with the algorithm because it might still be possible that we detect a syntactic difference elsewhere.

The third situation is when we find a variable on the left. Then we gather the

information that the subtree on the right at the corresponding position should be substituted for this variable. Since this variable can appear multiple times, it can happen that two or more term trees would have to be substituted for this variable. Even in this case it is too early to report that the rule should be rejected. It might happen that the multiple term trees to be substituted for a single variable could in fact be unified so that — given appropriate values for the variables occurring in these multiple terms — it could still be possible to apply the rule, which is exactly the situation called clog. If in this final stage no clog is detected we reject the rule if there are multiple terms for a single variable. Otherwise, the rule should be applied and we have constructed the necessary substitution in the meantime.

Our approach is practical because checking for unification is too costly and what we have is good enough because rewriting terms with variables (symbolic computation) is not meant to be intelligent at this stage (it would have to know all of logic) and it only has to serve in certain situations where we want to inline simple function calls and we need to symbolically rewrite terms with variables to achieve this.

The Rewriting Algorithm (Algorithm 4) given below takes two typed terms as input: l and t. The first one is interpreted as the left-hand side of the rewrite rule which we are trying to apply to the term t to be rewritten. We assume that t has no function names except at the top position and that l has term variables only in the leaves.

This algorithm returns

- REJECT when the rewrile rule cannot be applied and other rewrite rules should be tried;
- CLOG when the rewrite rule cannot be applied but because of a potential possibility that it could be applied the process of trying other rules must be stopped;
- APPLY when the rewrite rule can and should be applied, in which case the algorithm also returns a substitution s such that $\overline{s}(l) = t$.

The following easy example demonstrates the possibility of wrongly reporting clog when in fact there is none. Let $l = f(C(1, x), C(2, y))$ and $t = f(z, z)$, where f is a function name, C is a constructor, and x, y, z are term variables. It is easy to see that according to the mathematical definition of clog there is no clog here, but the algorithm — when it reaches the point of comparing $C(1, x)$ with the left-hand z — will report a clog.

Function names and term variables allowed

Finally, we will discuss the situation when our term to be rewritten contains function names. Let us demonstrate the problem with an example. Suppose we have these two rewrite rules:

(1) $equals(x, x) \to true$ with $vtype(x) = \alpha$,

(2) $equals(x, y) \to false$ with $vtype(x) = vtype(y) = \alpha$.

Algorithm 4 Rewriting Algorithm

STEP ONE
Let $SUBST = \emptyset$ and let $SubstVar = \emptyset$.
Let W be the set of positions that appear in both term trees l and t. We will be traversing W recursively from node to children so that we never visit a position when some position above has not been visited yet.
for all $p \in W$ **do**
  Let a be the subtree of l at position p.
  Let b be the subtree of t at position p.
  if a is a variable **then**
    $SUBST := SUBST \cup \{a \leftarrow b\}$
    $SubstVar := SubstVar \cup \{a\}$
  else if the top symbol of b is a variable **then**
    report CLOG and continue but abandon all positions below p
  else if the top symbols of a and b are different **then**
    return REJECT and stop
  end if
end for
if CLOG has been reported **then** return CLOG and stop

STEP TWO
for all $x \in SubstVar$ **do**
  run the **SUB ALGORITHM** on the set of term trees $\{\omega \colon x \leftarrow \omega \in SUBST\}$
  if it returns REJECT **then**
    return REJECT and stop
  else if it returns CLOG **then**
    report CLOG and continue
  end if
end for
if CLOG has been reported **then** return CLOG and stop
otherwise return APPLY and $SUBST$

SUB ALGORITHM for the set of term trees $\{\omega_1, \ldots, \omega_n\}$
if there is only one element in the set **then**
  return PASS
else if some $\omega_i = A(\ldots)$ and $\omega_j = B(\ldots)$ where $A \neq B$ are constructors **then**
  return REJECT
else if $w_1 = A(s_1^1, \ldots, s_k^1) \wedge \ldots \wedge w_n = A(s_1^n, \ldots, s_k^n)$ and A is a constructor **then**
  run the SUB ALGORITHM separately for the following k sets $\{s_1^1, \ldots, s_1^n\}, \ldots, \{s_k^1, \ldots, s_k^n\}$
  if there is at least one REJECT **then** return REJECT **else if** there is at least one CLOG **then** return CLOG **else** return PASS
else
  return CLOG
end if

Let f and g be two function names and let $equals(f(z), g(z))$ with $vtype(z) = int$ be the term which we want to rewrite. Naturally, the first rule cannot be applied and there is no clog. So according to what was said earlier, the system would have to move on to the second rule — which can be applied — and would have to rewrite the term into $false$, which is bad, because it might be possible that, say, $f(1) = g(1)$. So we need to introduce a special awareness into the system that will take care of such situations — which will treat function calls as unknowns, just like variables.

When we have a term with function names below the top position we must temporarily replace all function calls with term variables in a way which is a one-to-one correspondence. Then we treat such a modified term tree with the rewriting algorithm — which will be able to detect a clog in our example — and then we replace the temporary variables back with the original function calls.

In our example, it would be like this: the original term tree $equals(f(z), g(z))$ would be replaced with, say, $equals(tmpvar1, tmpvar2)$ and according to the rewriting algorithm there would be a clog detected and it would not be rewritten and then it would be replaced with $equals(f(z), g(z))$ and thus the rewriting procedure would be finished — leaving the term unchanged.

And $equals(f(z), f(z))$ would be replaced with $equals(tmpvar1, tmpvar1)$ and this time the rewriting algorithm would determine that the first rule can be applied and it would be rewritten into $true$. Since the temporary variables are no longer present, this is the final form of the original term after rewriting.

6.2 Normalisation

In the previous sections we have discussed the situation when we have a term with a function name at the top position and a set of rewrite rules which have the same function name at the top position on the left side. We have conclusively answered the question whether any of those rewrite rules should be applied to the term and if so, which of them should be used.

Now, we are prepared to discuss a more complex situation. We have a term and a whole set of rewrite rules with various function names at the top position. The question now is at which position to apply a rewrite rule. Notice that once we determine the position we already know from previous sections which rule to choose or whether to leave the term unchanged.

Normalisation is a process of rewriting a given term step by step until it cannot be further rewritten.

During this process it may happen that we encounter two identical subterms, in which case it might be a waste of time to normalise each of them separately and it might be useful to store the results. Such mechanism is called *memoisation* and we discuss it in more detail later. For now we are going to use MEMOISE as a special function that decides if a result should be kept in memory or not and we will use RETRIEVE as a function that gets a result from memory if it is there.

The set of function names has a special subset of *special function names*. Terms with special function names at the top position are not to be rewritten according to the algorithm presented above. To rewrite such a term is a different process which we will not describe at this moment. The important thing to keep in mind is that when we talk of applying REWRITE at a given position of a term we use the algorithm described above for normal functions and we use the special process — hitherto undescribed — for special functions. Moreover — in some cases we want to rewrite only the special functions and not the normal ones. We denote such rewrite function by REWRITE-SPECIAL and the appropriate normalisation procedure that uses always REWRITE-SPECIAL instead of REWRITE is called NORMALISE-SPECIAL.

We will describe an algorithm which takes a term and rewrites it until it cannot be further rewritten. It is possible that this algorithm will loop endlessly because of some inappropriate rewrite rules like for example: (1) $f(x) \to g(x)$ and $g(x) \to f(x)$, or (2) $f(x) \to f(f(x))$.

The normalisation algorithm start by looking for all positions in the input term at which there is a bracketing function that in speagram has the name `F\lb_\?_\rb` All such positions are normalised prior to all other, i.e. the brackets are removed.

Then the algorihtm performs the actual normalisation and rewriting with special care for the *if-then-else* function, which in speagram is named `Fif_\?_then_\?_else_?`. During this process we omit all subterms that have the *verbatim* function in the head, and *verbatim* is named `F<_|_\?_|_>`. This part of normalisation is presented as Algorithm 5 and you should note that whenever we refer to recursive normalisation in the algorithm we mean only this part of normalisation and not the whole process.

Algorithm 5 Normalisation Algorithm — Main Part

NORMALISE for $t = \psi(t_1, \ldots, t_k)$

if ψ is a constructor or a variable **then**
  let s_i be the recursively normalised term t_i and return $\psi(s_1, \ldots, s_k)$
else if ψ is *verbatim* **then**
  return t
else if ψ is *if-then-else*, $t = \psi(t_1, t_2, t_3)$ **then**
  let s_1 be the recursively normalised condition t_1 and let $t' = \psi(s_1, t_2, t_3)$
  if REWRITE $t' = t'$ **then**
    NORMALISE-SPECIAL t'
  else
    NORMALISE t'
  end if
else
  let s_i be the recursively normalised term t_i and $t' = \psi(s_1, \ldots, s_n)$
  if we can RETRIEVE t' **then** return the retrieved result
  if REWRITE $t' = t'$ **then** let $s = t'$ **else** let $s =$ NORMALISE (REWRITE t')
  MEMOISE the pair (t', s) if necessary and return s
end if

Finally, when the main part of normalisation is done, we get back to the *verbatim* functions and remove them according to the rule $verbatim(x) \to x$ from all positions above which there is no other verbatim.

In current implementation there is one more special case which prevents function normalisation when a new priority rewrite rule is added. It will probably be changed in the future so we do not describe it here.

6.2.1 Memoisation

We introduce a memoisation mechanism so that the system recognizes whether a given term has already been normalised before and immediately replaces its multiple occurrences with the already calculated normalised form.

You have to be aware that memoisation can improve performance a lot or even make some mistakenly exponential algorithms run in polynomial time, but memoising everything you encounter would take too much memory and decrease performance due to continuous hashtable lookups. Current implementation uses a heuristic to cope with this problem. Below is a rough sketch of a solution that we might implement in the future.

First of all we have to count the number of rewriting and memory lookup steps that were necessary to normalise a given term. When the number of steps is small we should *not* remember the result. This additionally has to depend on the linearity of the rewrite rule that was used — if the rule is non-linear then only a small number of steps can be allowed without normalisation, but if the rule is linear then we can allow quite a few steps. Linearity measurement has to take care of small constants in one parameter, since if it is the only non-linear thing then perhaps we do not need to memoise.

To make the above more precise we suggest the following strategy. We can distinguish three kinds of rules — linear ones, ones that are not linear but the size of non-linearity is 1 (e.g. `map`, `exists` and all other that are linear but for a function argument) and the other — higher non-linear ones. For each kind we should have a constant and if the number of rewriting steps done without memoisation exceeds the constant then we should memoise.

The additional possible improvement for memoisation is counting the hit-count of our cache (memoised stuff) and when some bound is reached we should reclaim all cache that has not been hit. At such point we can as well increase the numbers of rewriting steps done without memoisation, so that by long computations memory usage does not grow too fast.

Of course additionally memoisation needs to be customised. There should be a built-in function `memoise handler [function name as string]` which would return a handler — function that handles memoisation for the given name. The handler should take as arguments the term that is just being rewritten and the number of steps that elapsed and return a ternary truth value — true meaning that we should memoise, false meaning that we should not and unknown meaning that the default system

strategy should be applied. We should then recognise if a constant true or false or unknown value is set and then be able to use is efficiently. This will allow fast and efficient switching on and off memoisation for selected functions.

Chapter 7

Parser

7.1 Lexer

The lexer splits a given input string in such way that white spaces serve as delimiters and do not appear on the resulting list of tokens, and non-alphanumeric characters become separate tokens of length one. UTF starting characters (ASCII code above 127) are treated in the same way as alphanumeric ones.

There is one exception to this rule. If the lexer encounters a string `" whatever "` then it does not split it unless the first quote is directly prefixed by an ampersant. In such case the ampersant is removed.

You should as well note that the lexer performs standard XML syntax conversion for ampersant, quote, apostrophe and $<$ and $>$ at the start, so for example `&apos;` is read as a single `'` token and `&":::":&:"` is finally represented as `" : : " :&: "`.

Additionally in the end if the first word starts with an upper-case letter but it is not all upper-case then the first letter is changed to lower-case. The first word is defined as the first split string that is not an apostrophe, so `''Book` will be read as `''book`.

7.2 Definition of the Parser

Let K be the set of all nonempty finite sequences of tokens (interpreted as all possible results of lexing an input string). Let T be the set of all typed terms. Let S be a set of syntax definitions. We will define the parser by an inductive construction of the relation $\mathbb{P} \subset K \times T$ such that for every $(k,t) \in K \times T$ it holds that $(k,t) \in \mathbb{P}$ if and only if the term t is a correct result of parsing the sequence of tokens k by using syntax definitions from the set S. Such a t need not be unique.

For the definition of the parser we need a special type $string \in \mathcal{G}$ and a function `code_string` which takes a string and returns a typed term of type *string*. This function establishes a one-to-one correspondence between the set of all strings and the set of all terms of type *string*. It is a way of coding strings as terms.

Additionally, we need a special constructor `term_type_cons_name` and a special func-

tion `code_list` which takes a list of typed terms (possibly empty) and returns a typed term which uniquely encodes this list of terms.

The relation $\mathbb{P}$ is defined as the smallest set satisfying the following three conditions:

(1) If $k = (k_1, \ldots, k_N)$ is a nonempty sequence of tokens,
t is a typed term of height one,
and (a, b, c) is a syntax definition such that

(i) $a \neq$ type and $t = head()$

(ii) $\texttt{GeneratedName}(a, b, c) = head$

(iii) if $head$ is a term variable then $vtype(head) = c$

(iv) $b = (a_1, \ldots, a_N) = k$

then $(k, t) \in \mathbb{P}$.

(2) If $k = (k_1, \ldots, k_N)$ is a nonempty sequence of tokens,
t is a typed term, and (a, b, c) is a syntax definition such that

(i) $a =$ type and
$t = \texttt{term\_type\_cons\_name}(\texttt{code\_string}(head), \texttt{code\_list}([\,]))$

(ii) $\texttt{GeneratedName}(a, b, c) = head$

(iii) if $head$ is a term variable then $vtype(head) = c$

(iv) $b = (a_1, \ldots, a_N) = k$

(3) If $k = (k_1, \ldots, k_N)$ is a nonempty sequence of tokens,
t is a typed term, and (a, b, c) is a syntax definition such that

(i) if $a \neq$ type then $t = head(t_1, \ldots, t_M)$

(ii) if $a =$ type then
$t = \texttt{term\_type\_cons\_name}(\texttt{code\_string}(head), \texttt{code\_list}([t_1, \ldots, t_M]))$

(iii) $\texttt{GeneratedName}(a, b, c) = head$

(iv) if $head$ is a term variable then $vtype(head) = c$

(v) $b = (a_1, \ldots, a_K)$, where $M \leq K \leq N$ and each a_j is either a string or a type

(vi) there exist positive integers $i_1, \ldots, i_K$ and $r_1, \ldots, r_K$ such that
$i_j \leq r_j$ for all $1 \leq j \leq K$,
$r_j < i_{j+1}$ for all $1 \leq j < K$, and
$(1, 2, \ldots, N) = (i_1, \ldots, r_1, i_2, \ldots, r_2, i_3, \ldots, r_3, \ldots\ldots, i_K, \ldots, r_K)$

(vii) there exists a function α which is an increasing bijection from the set of all those j's such that a_j is a type to the set $\{1, 2, \ldots, M\}$

(viii) if a_j is a string then $i_j = r_j$ and $a_j = k_{i_j}$

(ix) if a_j is the *string* type (that is $a_j = string \in \mathcal{G}$)
then $i_j = r_j$ and $t_{\alpha(j)} = \texttt{code\_string}(k_{i_j})$

(x) if a_j is a type other than *string* then $((k_{i_j}, \ldots, k_{r_j}), t_{\alpha(j)}) \in \mathbb{P}$ and `TermType`$(t_{\alpha(j)})$ unifies with a_j

then $(k, t) \in \mathbb{P}$.

In practice, before we start parsing, we may filter the set of currently declared syntax definitions so that we only keep those which might be used and discard those which cannot be used.

7.3 Parsing Algorithm

We use a chart-based bottom-up parsing algorithm very similar to the one for context-free grammars (where types play the role of non-terminals). The additional step is checking if a resulting term can be well-typed and what is the corresponding type reconstruction. This is done each time when a syntax definition is fully applied to a sequence. You can refer to the article *Functional Pearls: Functional Chart Parsing of Context Free Grammars* by Peter Ljunglöf to read about chart-based parsing.

7.4 Disambiguation after Parsing

After parsing it might happen that there are multiple results and we have to use a mechanism for disambiguation to choose one of them.

First of all, if we have two distinct terms u and v as parsing results such that u matches v and v does not match u ($= u$ is effectively more general than v) then we remove the less general v from consideration — this is the first stage of disambiguation.

Secondly, we use a user-defined function `parse preferred to` which takes two terms t_1, t_2 and returns $1, -1, 0$ when respectively the first term is better, the second term is better or the terms cannot be compared. With the help of this function we create a comparison matrix for the results of parsing and we filter out all the results for which there is a better one.

The preference function is not built into the system. It is entirely defined in the Speagram library. Moreover, it is continually redefined by the user who writes his own Speagram code and can add new rules to this function, for example by using a macro like `see` $(x * y) + z$ `preferred to` $x * (y + z)$.

Appendix A

Conventions

A.1 Ocaml Coding Conventions

This appendix describes coding conventions, which are used while writing code for the Speagram project in Ocaml. As the project code can be changed by a few persons and we want to keep it readable you are kindly requested to keep your code consistent with them. In this way it is easier to read the code and introduce modifications.

You should note that one basic rule that you must adhere to when changing speagram code is that the code is and should be written in *clear and simple functional style* whenever possible. We sometimes use imperative constructs but functional ones are strongly preferred in all situations when it is not clear that the imperative code is much better suited. We *do not* use object-oriented constructs in speagram code and *all defined types are algebraic*. As result there are no mutable values used in Speagram code except for array and hashtable elements and specifically declared *ref*erences. This is very comfortable because it allows us to apply changes to elements without worying that we are breaking something else somewhere far away in the code at the same time

A.1.1 Files, Headers and Environment

Project files should be put in `src/` directory. Names of files with modules should be written in CamelCase where only separate words start with uppercase letters, e.g. `BuiltinLang`. Names of executable files should be written in lowercase with an underscore "_" being used as a separator (i.e. space).

All lines in a file should have at most 79 characters. This is because wrapping too long line leads to unpleasant look, and turning wrapping off causes part of the line to be off-screen. Indentation is defined as 2 spaces.

A source file should begin with a comment containing: a line with project name (in this case: `Speagram`), an empty line, a copyright line, an empty line, and then the BSD license. Additionally, the comment should contain a short description of what is in the file. This short description should be appended to the file comment as: an empty line, a dashed (-) line, an empty line and then a short description of what is

in the file.

After the first comment we leave two blank lines and then open all used modules. We first open Ocaml standard library modules, then leave a blank line and open Speagram modules. After that we put commented-out lines that load these modules since this is useful when working with ocaml interpreter. Basic example of opening and loading modules is the following. Please note and respect the way the *load* statement is commented out, as precisely this way makes it easy to uncomment it and comment it back again fast.

```
(* opening comment with licence and short description *)


open Array;;
open Str;;

open Type;;
(* * #load "Type.cmo";; *)
```

We try to use only the standard Ocaml environment and avoid external modules as this makes the build process complex. But we normally link to the `Str` library and use the CamlP4 preprocessor, so if you want to start Ocaml interpreter to play with Speagram cose you should use the following (under linux and with standard paths)

```
ocaml -I +camlp4 camlp4o.cma str.cma
```

A.1.2 Code Layout in Files

We do not put strict restrictions on code layout in our files, you should try to stick to the standard ocaml conventions and look at the code and the example below. Please *do not* use CamelCase in code, we use lowercase names and underscore "_" as space inside names.

Still we have a preferred way of organising the file structure. We usually start the file with the definition of new algebraic types as necessary and function definitions follow later.

It is advisable to use clear separators of areas of functionality inside the file. To achieve this, use the following comment scheme:

```
(* -------- FOLLOWING FUNCTIONS DO THIS AND THAT ------------- *)
```

so that the last dash is at the 79th position on the line (this makes all such comments more visible).

We often put short tests inside the code as many bugs get detected early in this way and when working with an interpreter we run these tests to check if nothing was broken. Tests are put near the functions that are tested and are in a block that is commented out but easy to uncomment as depicted below.

```
(* TESTS *
   test1
   test2
*)
```

We normally put one blank line between such separator and the next definition and between related definitions and two blank lines between less related definitions and before the next separator. Normally comments are put before each non-local function, as depicted below. Of course you should use meaningful names for functions and variables and write useful comments, not like the ones below.

```
(* -------- FOLLOWING FUNCTIONS DO THIS AND THAT ------------- *)

(* Technical function that computes this and that detail. *)
let fu_fu_fu x =
  let fi_fi_fi s = ... in
  fo_fo_fo (fi_fi_fi x)
;;

(* Another more important function that uses fu_fu_fu. *)
let imp_imp x = ...;;

(* TESTS *
   fu_fu_fu test;;
   imp_imp test;;
*)


(* ------------ NEXT PART AND ANOTHER FUNCTIONS --------------- *)

let another_fun z =
  ...
;;
```

A.2 Documenting Conventions

In general we adhere to the standard LaTeXconventions for chapters, sections, subsections and other things. Small sections should be started with the special `\smallsection` command and do not forget to use hyphens and dashes in the correct way, so you should distinguish - and – and —.

We try to keep TeX-files clean, lines up to 79 characters long, new sentences starting in new lines and we define all additional LaTeX-functions in the main `speagram.tex` file.

You should remember that TeXmakes it possible to define the layout according to the semantics (meaning) of the text. Therefore if you have a part of text that you

want to layout in a special way you should not enforce layout directly but rather try to define special function for such messages, define it in the main file and use it. Especially you should not use the `\it` and `\bf` commands to make text bold or italic, you should define a special function for your text or use `\em` if you just want to emphasize something.

There are a few smaller technical issues that we assume are used throughout the documentation and relate to the predefined commands we use.

First of all we use enumerate and url packages and we normally enumerate using small Roman or Arabic digits in brackets, so we use the following for enumeration.

```
\begin{enumerate}[(1)] or [(i)]
\item ...
\end{enumerate}
```

For url references we use the following.

```
\url{http://www-address}
```

We open the mathematical environment with one of the following, according to the situation and using the simplest one that looks well.

```
$ maths $ - very simple
$$ maths $$ - simple
\begin{align} maths \end{align} - with numeration
```

We also have a few predefined commands that we use in corresponding situations as we consider them more readable or just shorter than the standard LaTeXequivalents.

- `\func` to refer to functions in source code,
- `\class` to refer to classes in source code,
- `\codepath` to write paths to directories,
- `\code` to refer to other parts of source code,
- `\smul`, `\ssum`, `\sminus` and `\sbigmul`, `\sbigsum` for set multiplication ($\cap$), addition ($\cup$) and subtraction ($\setminus$), also in bigger versions,
- `\lor`, `\land`, `\lbigor`, `\lbigand` for logical or ($\vee$) and and ($\wedge$), also in bigger versions,
- `\to` ($\rightarrow$), `\ot` ($\leftarrow$), `\implies` ($\Longrightarrow$), `\iff` ($\Longleftrightarrow$) for useful arrows,
- `\forall` ($\forall$), `\exists` ($\exists$) for quantifiers.

We also use shorter versions of commands to change the style of mathematical text:

```
\mr{text} for straight (roman),
\mc{text} for caligraphic,
\mbb{text} for math blackboard bold,
\mfr{text} for fracture.
```

You can see here how "A" looks in normal maths A, straight A, calligraphic $\mathcal{A}$, math blackboard bold $\mathbb{A}$ and fracture $\mathfrak{A}$. We normally use straight text for functions that are defined and have more than one letter in the name, like min or max and the other styles according to usual mathematical conventions that make us denote the power set of natural numbers by $\mathcal{P}(\mathbb{N})$.

For presenting algorithms we use the `algorithmic` package and environment and inside this environment we use `\textbf` freely, as opposed to other parts of the document.