Ben–Gurion University of the Negev,
Department of Mathematics & Computer Science

CFUF: A Fast Interpreter for the Functional Unification Formalism

Thesis submitted as part of the requirements for the M.Sc. degree of Ben–Gurion University of the Negev

by
Mark Kharitonov
(mark@cs.bgu.ac.il)

The work for this thesis has been carried out at Ben–Gurion University of the Negev,
under the direction of Dr. Michael Elhadad

April 18, 1999
Subject:

CFUF: A Fast Interpreter for the Functional Unification Formalism

This thesis is submitted as part of the requirements for the M.Sc. degree

Written by: Mark Kharitonov
Advisor: Dr. Michael Elhadad
Department: Math & Computer Science
Faculty: Exact Sciences
Ben–Gurion University of the Negev

Author signature: _______  Date: ______
Advisor signature: _______  Date: ______
Dept. Committee Chairman sig.: _______  Date: _____
I would like to thank:

- Michael Elhadad, the advisor of this work
- Yael Netzer-Dahan, the first beta-tester of the CFUF system
- Parents and Family.

Mark Kharitonov
Abstract

Functional Unification is a popular formalism for the implementation of natural language generation (NLG) system. The FUF programming language is a particular implementation of this formalism that is widely used within the NLG community. I present in this work the design, implementation and evaluation of CFUF, a fast interpreter for the FUF language.

CFUF is fully compatible with the existing reference implementation of FUF that has been available since 1990. It can, in particular, run unchanged the SURGE generation grammar of English which has been developed in FUF.

CFUF is built on the model of a virtual machine for the execution of FUF programs. The overall process of execution in CFUF is to compile input FUF terms (called functional descriptions or FDs) into an internal graph representation. The set of commands to manipulate such graph representations (FDC) is the base language of the FDVM (FD Virtual Machine).

FUF grammars are translated into an AND–OR tree where each and–node contains a sequence of FDC commands. The FUF interpreter traverses the AND–OR tree in a non–deterministic manner and executes the FDC commands in the right context (as encoded in the set of registers of the FDVM). It also handles backtracking.

The design of the FDVM takes into account the empirical fact that FUF programs are extremely non–deterministic and include lots of backtracking. We have therefore favored in the design decisions fast handling of backtracking and undoing. Particular attention has been paid to avoid copying structures in memory as well, which is known to cost a lot of the practical runtime of unification–based formalisms. The result is a design based on a "clock mechanism" where modifications to the FD graph are never undone, but simply made invalid when a clock is advanced. Traversal of the graph is made, as a result more expensive, but the tradeoff is shown to be beneficial.

Detailed performance evaluation is provided that demonstrates the well–foundedness of the major design decisions. An implementation of the CFUF design is provided and compared with the existing FUF interpreter. On large inputs, CFUF proves to be close to 100 times faster than the existing interpreter.
# Contents

1 Introduction 1

2 Background 4
   2.1 Motivation .......................... 4
   2.2 How FUF is Used .......................... 5
   2.3 Partial vs. Complete Specification ............... 6
   2.4 Unification Process .......................... 7
   2.5 FD/Grammar BNF .......................... 7
   2.6 A Short Example .......................... 9
   2.7 Paths, Types and Special Features ............... 11
      2.7.1 Paths .................................. 11
      2.7.2 Types .................................. 13
      2.7.3 Special Features ......................... 15
   2.8 Existing Implementation .......................... 17
   2.9 Objectives of this Work .......................... 17
   2.10 Summary .................................. 17

3 FD Translation 19
   3.1 Overview .................................. 19
   3.2 Intermediary Graph .......................... 20
   3.3 Quotient Graph .............................. 23
      3.3.1 Node Merger Implementation ............... 25
      3.3.2 Node/Arc Validity .......................... 30
   3.4 The Translation Process .......................... 31
      3.4.1 FD Parse Tree Traversal .................... 31
      3.4.2 FD Virtual Machine (FDVM) ................ 35
   3.5 Summary .................................. 36
4 Grammar Translation 38
4.1 Overview .................................. 38
4.2 Grammar Tree .............................. 38
4.3 Summary .................................. 42

5 Special features 44
5.1 Overview .................................. 44
5.2 Fset ....................................... 44
5.3 Cset ....................................... 45
5.4 Pattern .................................... 47
5.4.1 Patterns and Grammar .................. 54
5.5 Summary .................................. 55

6 Unification 57
6.1 Overview .................................. 57
6.2 The Current Grammar Branch Selection 59
6.3 The Current Constituent Selection .......................... 60
6.4 Constituent–Branch Unification .......................... 64
6.5 Backtracking ................................ 65
6.5.1 Node type ................................ 66
6.5.2 Node Link ................................ 68
6.6 Summary .................................. 69

7 Evaluation 70
7.1 Overview .................................. 70
7.2 CFUF vs LispFUf ............................ 71
7.3 CFUF Profiling .............................. 75
7.3.1 Type Sequence ............................ 79
7.3.2 Link Sequence ............................ 82
7.3.3 Conclusion: Average Cost of Type/Child Lookup in
the Presence of Backtracking ..................... 96
7.3.4 Composite Nodes Size/Dead Nodes Percentage .... 97
7.4 Impact of Index .............................. 98
7.5 Summary .................................. 99
8 Implementation Issues \hfill 100
  8.1 Overview \hfill 100
  8.2 Interpreter \hfill 101
  8.3 Memory Management \hfill 102
  8.4 Containers \hfill 102

9 Conclusion \hfill 103
List of Tables

2.1 Operator $\sqcup (t_1, t_2 \in T)$. .................................................. 14

3.1 FDC commands .......................................................... 20
3.2 Node operations ......................................................... 29
3.3 CN register .............................................................. 35
3.4 FDC commands (all, but AddPath) ................................. 37

4.1 ((alt (branch$_1$ branch$_2$ ... branch$_n$))) ......................... 40
4.2 ((alt (...)) (alt (...)) ... (alt (...))) .............................. 41

7.1 CFUF largest times on the input set (on Linux i586) ............ 73
7.2 Nesting times (on Linux i586) ........................................ 75
7.3 CFUF greatest times (on Solaris sparc Ultra-4) .................. 76
7.4 Nesting times (on Solaris sparc Ultra-4) .......................... 77
7.5 The largest values of $RV_{tmf}(2)$. Indicates the probability that
0, 1 or 2 type tags must be checked until the valid type of a
node is found. ............................................................. 80
7.6 The smallest values of $RV_{tmf}(0)$ .................................. 80
7.7 The largest values of $RV_{lsuf}$: Probability that a valid link is
found in 1, 2, 3 or more clock tag checks ............................. 84
7.8 The smallest values of $RV_{lsuf}(0)$. ................................. 84
7.9 The largest values of $RV_{lsuf}(i)$ ................................... 86
7.10 The greatest values of $RV_c(6)$ ..................................... 90
7.11 The smallest values of $RV_c(1)$ ................................... 90
7.12 The greatest values of $RV_{af}(3)$ .................................. 93
7.13 The smallest values of $RV_{af}(1)$ .................................. 94
7.14 The largest values of $RV_n(i)$ ..................................... 98

iv
List of Figures

2.1 A trivial semantic formula .............................. 6
2.2 A simple FUF grammar gr0 ............................ 9
2.3 A simple fd ir0_1 .................................... 10
2.4 Result of unifying ir0_1 with gr0 ..................... 11
2.5 Trace of the process of unification ir0_1 with gr0 .... 12
2.6 Path example ......................................... 13
2.7 Special feature examples .............................. 16

3.1 Merger example ....................................... 23
3.2 follow((a, c), (1, (d))) \not\iff follow((b, c), (1, (d))) .... 25
3.3 Node merger ......................................... 28
3.4 Node/Arc removal ................................... 30
3.5 FD translation scheme ................................ 31
3.6 FD translation scheme ................................ 31
3.7 A parse tree traversal example ...................... 33
3.8 The quotient graph: from \alpha_1 to \alpha_7 ............ 34

4.1 The scheme of the grammar tree ...................... 40
4.2 A trivial grammar .................................... 42

5.1 Distinction feature vs. constituent: tense is not a constituent 46
5.2 Using cset to distinguish sub-constituents from other features 47
5.3 Specifying the order of sub-constituents with pattern .... 48
5.4 Patterns as grids .................................... 52
5.5 Set of patterns as grid ................................ 53
5.6 A possible 3-dimensional grid for (\delta, n_i, \delta, m_i, \delta) .... 53
LIST OF FIGURES

6.1 The procedure to identify the subconstituents of the current constituent \( N \) .................................. 62
6.2 The scheme of constituent selection .................................................. 62
6.3 Unification example with the given type ........................................... 63

7.1 LispFU vs. CFUF (on Linux i586) X-axis indicates the number of backtracking points, Y-axis the speedup factor CFUF vs. LispFU ................................................................. 72
7.2 Nesting: LispFU vs. CFUF (on Linux i586) X-axis is the number of backtracking points, Y-axis is speedup factor. Benchmark is on the set of nested clauses “I think that ...” ........ 74
7.3 LispFU vs CFUF (on Solaris sparc Ultra-4) X-axis is the number of backtracking points; Y-axis is the speedup .... 76
7.4 Nesting: LispFU vs. CFUF (on Solaris sparc Ultra-4) X-axis is the number of backtracking points; Y-axis is speedup. . 77
7.5 \( RV_{lsuf}(0) \): Probability that the first type checked is the valid type ........................................ 81
7.6 \( RV_{lsuf}(1) \): Probability that two type checks are necessary to find the valid type ..................... 82
7.7 \( RV_{lsuf}(2) \): Probability that 3 type checks are necessary to find the valid type .............................. 83
7.8 \( RV_{lmuf}(0) \): Probability that the valid link is found in 1 clock tag check (optimal case) .................. 85
7.9 \( RV_{lmuf}(1) \): Probability that a valid link is found in 2 clock tag checks ........................................ 86
7.10 \( RV_{lmuf}(2) \): Probability that a valid link is found in 3 clock tag checks ...................................... 87
7.11 \( RV_{seq}(1) \) .................................................................................. 87
7.12 \( RV_{seq}(2) \) .................................................................................. 88
7.13 \( RV_{seq}(3) \) .................................................................................. 88
7.14 \( RV_c(1) \) ..................................................................................... 91
7.15 \( RV_c(2) \) ..................................................................................... 92
7.16 \( RV_c(3) \) ..................................................................................... 92
7.17 \( RV_c(4) \), \( RV_c(5) \) and \( RV_c(6) \) ........................................................ 93
7.18 \( RV_{uf}(1) \) ..................................................................................... 95
7.19 \( RV_{uf}(2) \) ..................................................................................... 95
7.20 \( RV_{uf}(3) \) ..................................................................................... 96
Chapter 1

Introduction

There are a number of formalisms used in Natural Language Generation (NLG) research. One of them, called Functional Unification Formalism (FUF), was devised and implemented by Elhadad [3]. LispFUF (the original Lisp implementation of FUF) is an interactive system, manipulating FDs and grammars specified by the user from the interpreter prompt.

LispFUF turned out to be very convenient for NLG research and it has been widely distributed within the NLG community. In a recent workshop on generation (Summer 1998 [6]) for example, more than half of the practical system discussed were using the FUF system and its associated grammars for syntactic realization. The main disadvantage of FUF with respect to other syntactic realization systems has been its execution speed.

This work addresses this issue. It presents the CFUF interpreter of the FUF language. CFUF is an implementation of FUF, which combines the power and usability of LispFUF with much improved execution speed. In particular, using the SURGE generation grammar of English [5] which has been developed in FUF, CFUF can generate full paragraphs in less than a second vs. one to two minutes for LispFUF on current PC-type hardware. This makes it possible to generate paragraphs in real-time for speech production systems using the wide-coverage and comfort of a large-coverage grammar like SURGE.

To meet this speed requirement, CFUF is built on the model of a virtual machine for the execution of FUF programs. The overall process of execution in CFUF is to compile input FUF terms (called functional descriptions or FDs) into an internal graph representation. The set of commands to manip-
ulate such graph representations (FDC) is the base language of the FDVM (FD Virtual Machine).

FUF grammars are translated into an AND–OR tree where each and-
node contains a sequence of FDC commands. The FUF interpreter traverses
the AND–OR tree in a non–deterministic manner and executes the FDC
commands in the right context (as encoded in the set of registers of the
FDVM). It also handles backtracking and context management.

The design of the FDVM takes into account the empirical fact that FUF
programs are extremely non–deterministic and include lots of backtracking.
We have, therefore, favored in the design decisions fast handling of backtracking
and undoing. Particular attention has also been paid to avoid copying
structures in memory, which is known to cost a lot of the practical runtime of
unification–based formalisms. The result is a design based on a “clock mechan-
anism” where modifications to the FD graph are never undone, but simply
made invalid when a global clock is advanced. Traversal of the FD graph is
made, as a result more expensive, but the tradeoff is shown to be beneficial.

Detailed performance evaluation is provided that demonstrates the well–
foundedness of the major design decisions. An implementation of the CFUF
design is provided and compared with the existing FUF interpreter. On
large inputs, CFUF proves to be close to 100 times faster than the existing
interpreter.

The rest of the thesis contains:

– General background on FUF: motivation, how it is used, examples,
principal components and syntax (2).

– The description of the FD translation: internal representation and FD
Virtual Machine (3).

– The description of the grammar translation by giving the internal rep-
resentation (4).

– The special features of FUF (cset, fset, pattern, control): definition
and unification rules (5).

– Unification of FD and grammar: overall picture and detailed discussion
on the four major components — current grammar branch selection,
current constituent selection, constituent–branch unification and back-
tracking (6).
• Evaluation of CFUF: CFUF vs LispFUF and CFUF profiling (7).
• Various implementation issues: design, interpreter, memory management (8).
Chapter 2

Background

This work presents an efficient implementation of a FUF interpreter. FUF is a special-purpose language created to implement text generation systems. In this section, I present the main aspects of the FUF language that make it appropriate for the task of text generation and that have motivated its design. I then present a very brief overview of how the FUF language is typically used. This includes a presentation of the FUF syntax and short examples.

2.1 Motivation

FUF was developed by Elhadad [3] as an implementation and extension of the FUG formalism [7] — Functional Unification Grammar. In FUG, all linguistic information (input and grammar) is stored in special structures which encode both functional and structural information.

FUF is an extension of FUG and includes more control features and type hierarchy motivated by the specific needs of generation: lexicalization, phrase planning and even non-linguistic purposes that have appeared during the development of practical systems.

According to Robin [11], the benefits of using FUF for developing generation systems (as opposed to a direct syntactic encoding of the linguistic information) are:

1. The formalism supports partial information, is declarative, uniform and compact.
CHAPTER 2. BACKGROUND

2. The same formalism can be used for all stages of the generation process — syntactic realization, lexicalization, text planning. This precludes the need for interfaces between stages which have been very common with other formalisms.

3. FUF, because it supports delayed evaluation, allows easy experimentation with different ways to distribute subtasks among components at implementation time.

A wide-coverage grammar for English was developed using FUF — SURGE [4], [5] and the FUF/SURGE package is used in many projects in NLG research.

Using the FUF/SURGE package, implementing a generation system consists of decomposing non-syntactic processing into sub-processes and encoding in FUF the knowledge sources for each of these sub-tasks (see [11] for a complete discussion).

From a linguistic perspective, the functional analysis of language (in contrast to a structural analysis), identifies the function of each substring within the larger string of words. The structural stage of this analysis identifies the constituents of a phrase. For example, within the clause John eats an apple, a functional analysis would identify the actor (John) and affected (an apple) roles. A structural analysis only, in contrast, would only identify the structure of the clause as a concatenation of NP VP NP sub-structures. Functional Unification as a formalism is strongly geared to a functional type of analysis. The FD is used to represent the functional analysis of a phrase ([8]).

The functional analysis suits the generation process, since the source to the text eventually generated is conceptual/semantic and refers to the function of the different concepts (before the mapping to the syntactic level).

2.2 How FUF is Used

FUF is used to translate semantic formulas (predicate-argument structures) to sentences.

Figure 2.1 shows a semantic formula which is translated to the sentence “John eats an apple.”.
From now on I will use the term Functional Description or FD when referring to such predicate–argument structures. In the next sections I demonstrate the main features of the FUF formalism and how they impact on the generation process.

2.3 Partial vs. Complete Specification

The semantic input is always underspecified. We could receive several sentences for the input in Figure 2.1. For example:

“An apple is eaten by John.” — because voice is not specified.

“John ate an apple.” — because tense is not specified.

“Did John eat an apple?” — because mode is not specified.

“John must eat an apple” — because modality is not specified.

At a deeper level, a semantic formula of that sort can never be completely specified, because it is relative to a grammar — being completely specified for a grammar A, does not mean so for a grammar B. The FUF formalism makes support for partial specifications a central concern. This goes further than even Prolog. In Prolog, a term can be underspecified if it has unbound variables, for example p(x, b, c). But in Prolog, the head of the term (p in the example) and the arity of the term cannot be left underspecified. Radically, in FUF, these 2 restrictions (fixed head and arity) are removed.

```
[ action = eat ]
[ actor = John ]
[ object = apple ]
```

Figure 2.1: A trivial semantic formula
2.4 Unification Process

Using FUF, a sentence is obtained by unifying the input FD with a grammar, also written as a FUF term. Overall, sentence generation proceeds according to the following intuitive schema:

The **Target Sentence** conveys some meaning under grammatical constraints.

The **Input FD** specification encodes this meaning (partially)

The **Grammar** encodes grammatical constraints (in other words, the syntax of the language).

The **Unification process** combines meaning and syntax.

The **Linearization process** produces a linear sentence (an ordered list of morphologically inflected tokens) from the combined FD returned by the unification.

Critically, both the input and the grammar are encoded in the FUF formalism. The only operation defined between FUF terms is unification. The precise form of unification (functional unification) used here will be detailed in the rest of the thesis. It is slightly different from the unification operation defined in Prolog.

The implementation of the FUF interpreter focuses on the construction and manipulation of FD terms and on the unification operation. A distinct mechanism is implemented to support linearization and the handling of linearization constraints.

2.5 FD/Grammar BNF

In order to present examples, I first introduce the formal syntax of FUF in BNF. As usual, the symbols $[ ]$ $<>?+*$ \ are used in describing the syntax, but are not part of it:

- square brackets denote one value exactly from the given set
- angle brackets denote a BNF non terminal
- question mark means at most one occurrence
• plus means at least one occurrence

• star means any number of occurrences (zero as well)

• backslash prepended to a symbol makes that symbol part of the syntax as is.

```xml
<feature> ::= atom
<path> ::= {[^0 1 2 3 4 5 6 7 8 9]*} atom*
<type> ::= nil | any | given | none
        | atom
        | 'string'
        | #((under sunder < <=] atom)
<cset> ::= (attr+)
        | ([= (attr*) (== (attr*)) (\+ (attr*)) (- (attr*))]+)
<pattern> ::= ([dots attr (* attr])+
<fset> ::= (atom+)
<attr> ::= <feature> | <path>
:value> ::= <type> | <path> | <fset> | <cset> | <pattern>
<fd expr> ::= (<attr> <FD>)
            | (fset <fset>)
            | (cset <cset>)
            | (pattern <pattern>)
<FD> ::= <type> | <path> | (<fd expr>*)
<alt name> ::= atom
<wait expr> ::= <path> | <feature>
              | ([<path> <feature>] <type>)
<alt param> ::= (:index [atom (atom+)])
              | (:bk-class [atom (atom+)])
              | (:wait [<wait expr> | (<wait expr>*)])
              | (:demo 'string')
<gr expr> ::= <fd expr>
            | ([alt ralt] <alt name>? <alt param>* (Grammar+))
            | (opt <alt name>? Grammar)
<Grammar> ::= <type> | <path> | (<gr expr>*)
```
2.6 A Short Example

Refer to Figures 2.2 and 2.3 for examples of a grammar and an FD.

FUF is used to unify FDs with a grammar. An FD represents a partial description of a phrase or a sentence in a natural language, while the grammar is a description of that language's syntactic rules. By unifying the two, we hope to enrich the original incomplete input with the information necessary to produce the intended phrase or sentence. For instance, the grammar gr0 (Figure 2.2) is a grammar of a very simplified English. Let us look at the breakdown of the toplevel alt (which is short for alternation):

| (cat s) | corresponds to the Sentence category. The branch it heads specifies that: “An English sentence consists of three major parts: prot, verb and goal in this exact order, prot and goal belong to the Noun Phrase category and verb — to the Verb Phrase category. Furthermore, prot and verb have matching number (either both plural or both singular)”.
| (cat np) | encodes rules about the Noun Phrase category. The branch it heads specifies that: “An English noun phrase can be either proper or not. A proper noun phrase consists of a noun only, as opposed to |
a non-proper one which has its noun prepended by the article. The number of the whole noun phrase is determined by that of the noun”.

**cat vp** — corresponds to the **Verb P**rase category. The branch it heads specifies that: “An English verb phrase is just a verb and nothing else.”

The remaining three categories (noun, article, and verb) are leaves (or terminals in CFG terminology).

Let us turn our attention to the interpretation of the input ir0_1. It is a partial specification of the sentence “John likes Mary” in the above simplified English. The input itself bears no indication on the order in which the words are printed, as well as the number, tense, etc... It is the job of the unification to try and enrich the input with the appropriate constraints taken from the respective grammar. This, of course, may yield more than one solution if the input is too underspecified or fail altogether if the input contradicts the constraints expressed in the grammar.

Figure 2.4 shows the result of unification ir0_1 with gr0 (the features added by the grammar appear in bold).

Figure 2.5 shows the trace of the unification process developed by FUF when unifying ir0_1 with gr0. Overall, the FUF interpreter traverses the grammar by selecting non-deterministically branches in each alternation of the grammar. When a branch is selected, the attribute-value pairs in the grammar are tested against those appearing in the input at the same level. If the pairs match, unification proceeds. If the pair appears in the grammar only, it is then added to the input FD. This insures that the input is enriched into an FD that combines information from the original input FD and from the parts of the grammar that are compatible with the original input.

In many ways, FUF is close to Prolog:

- Both have unification at their core
• A Prolog program is analogous to a FUF grammar.

• A Prolog query is analogous to a FUF input.

In Prolog, however, we deal with First Order Term unification, which is in effect unification of trees, while in FUF arbitrary rooted directed graphs get unified. The path construct is the one that makes it possible to create arbitrary graphs. In that sense, Prolog unification is a particular case of FUF unification, where neither the grammar nor the input contain path specifications.

2.7 Paths, Types and Special Features

Now that we have the global picture, let us take a closer look at the principal components of a FUF input — paths, types and special features.

2.7.1 Paths

The path construct makes it possible to specify arbitrary graphs as input to FUF. There are two kinds of paths — **absolute** and **relative** (much like Unix directory paths). An absolute path lists the features which must be
CHAPTER 2. BACKGROUND

Figure 2.5: Trace of the process of unification ir0_1 with gr0
Using absolute paths:

\[
((a \ b \ c \ e) \ (a))
\]

\[
((a \ b) \ (d)))
\]

Using relative paths:

\[
((a ((b \ \wedge \ \wedge \ d))
\]

\[
(b ((c ((e \ \wedge^3))))))
\]

Figure 2.6: Path example

followed from the FD root in order to reach the path destination and it has the syntax \{\text{feat}_1 \ \text{feat}_2 \ \ldots \ \text{feat}_n\}. A relative path consists of two parts. The first part indicates how much to ascend up from the current location and the second lists the features to be followed then. The syntax is \{\wedge \ \wedge \ \ldots \ \wedge \text{feat}_1 \ \text{feat}_2 \ \ldots \ \text{feat}_n\}, where the measure of ascent is determined by the number of \wedge. One can write \wedge m for \wedge^m. The list of features is allowed to be empty. Figure 2.6 has an example.

2.7.2 Types

Ordinarily an FD is a list of attribute–value pairs. The attribute is either a feature name, like \text{cat} in the above example, or it can be a path, if the value is to be assigned to other location. Value is either a path (when features have to share values) or a type. \text{Vp, np} and \text{noun} are examples of types.

There are 6 predefined types: \text{nil, any, given, none, in:nil} and \text{in:any}. Of these, the last two cannot be specified in the input: \text{in:nil} and \text{in:any} are for FUF internal use.

Let \text{T} denote the set of all the atomic\(^1\) non predefined types. \text{T} is arranged in a hierarchy (specified by the user) to impose a partial order \leq on the

\(^1\)There are three kinds of non predefined types: \text{atomic} like \text{np} and \text{vp}; \text{string types} like `'Mary'` and `'John'`; and \text{under types} like `#(under np)` (or `#(<= np)`) and `#(under clause)` (or `#(< np)`).
Table 2.1: Operator $\sqcup (t_1, t_2 \in T)$.

<table>
<thead>
<tr>
<th></th>
<th>nil</th>
<th>any</th>
<th>given</th>
<th>none</th>
<th>atomic $t_2$</th>
<th>string $str_2$</th>
<th>$#(&lt;= t_2)$</th>
<th>$#(&lt; t_2)$</th>
<th>nil</th>
<th>any</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>nil</td>
<td>any</td>
<td></td>
<td>none</td>
<td>$t_2$</td>
<td>$str_2$</td>
<td></td>
<td></td>
<td>m-nil</td>
<td>m-any</td>
</tr>
<tr>
<td>any</td>
<td>any</td>
<td>any</td>
<td></td>
<td>none</td>
<td>$t_2$</td>
<td>$str_2$</td>
<td></td>
<td></td>
<td>m-nil</td>
<td>m-any</td>
</tr>
<tr>
<td>given</td>
<td>any</td>
<td>any</td>
<td>$\top$</td>
<td>none</td>
<td>$t_2$</td>
<td>$str_2$</td>
<td></td>
<td></td>
<td>m-nil</td>
<td>m-any</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td></td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m-nil</td>
<td>m-any</td>
</tr>
<tr>
<td>atomic $t_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$\top$</td>
<td>none</td>
<td>$glb(t_1, t_2)$</td>
<td>$\bot$</td>
<td>$\ast$</td>
<td>$\ast$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>string $str_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
<td>$\bot$</td>
<td>$\ast$</td>
<td>$\ast$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$#(&lt;= t_1)$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td></td>
<td>$\bot$</td>
<td>$\ast$</td>
<td>$\ast$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$#(&lt; t_1)$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td></td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>m-nil</td>
<td>m-nil</td>
<td>m-nil</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>m-any</td>
<td>m-any</td>
<td>m-any</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

Types$^2$. Type $t_1$ is said to subsume type $t_2$ if $t_2 \subseteq t_1$. Another way to express the same thing, is to say that “$t_1$ is more general than $t_2$” or “$t_2$ is more specialized than $t_1$”. The $\bot$ type (indicates failure) is more specialized than any type: $\forall t \in T(\bot \subseteq t)$.

The definitions of several useful notions follow:

- $S(t) = \{ s \in T \mid s \leq t \}$ — the set all the types more specialized than $t$.
  $max(S \subseteq T) = \{ s \in S \mid \nexists r \in S(s \leq r) \}$ - the set of maximal elements of a set of types $S$.

- Greatest Lower Bound:
  Let $t_1, t_2 \in T$ then $glb(t_1, t_2)$ denotes the greatest lower bound of the types $t_1$ and $t_2$ and is defined as:

  $$glb(t_1, t_2) = \begin{cases} t_{12} & \text{if } max(S(t_1) \cap S(t_2)) = \{ t_{12} \} \\ \bot & \text{if } |max(S(t_1) \cap S(t_2))| \neq 1 \end{cases}$$

- Type consistency:
  Let $t_1, t_2 \in T$, then $t_1$ and $t_2$ are consistent if and only if $glb(t_1, t_2) \neq \bot$.

Let the operator $\sqcup$ be the unification operator, i.e., $t_1 \sqcup t_2$ denotes the unification result for any two types (including predefined) $t_1, t_2$. Table 2.1 shows the results of unification of various types (both predefined and non predefined). From now on, $T_{all}$ will denote the set of all the types.

---

$^2$Being a partial order, the relationship $\leq$ is reflexive, antisymmetric and transitive.
† The operation succeeds whether the non \textit{given} operand comes from the original input FD or from the grammar. There is another semantic in which the unification fails if the latter is true.
‡ See Type–Node consistency in section 3.

\[ \forall s, r \in T \#(\leq s) \sqcup r = \begin{cases} r & \text{if } \text{glb}(s, r) = r^3 \\ \bot & \text{otherwise} \end{cases} \]

\[ \forall s, r \in T \#(< s) \sqcup r = \begin{cases} r & \text{if } \text{glb}(s, r) = r \neq s^4 \\ \bot & \text{otherwise} \end{cases} \]

\[ \forall s, r \in T \#(< s) \sqcup \#(< r) = \begin{cases} \#(< r) & \text{if } \text{glb}(s, r) = r \\ \#(< \text{glb}(s, r)) & \text{if } r \neq \text{glb}(s, r) \in T \\ \bot & \text{otherwise} \end{cases} \]

**** Let \( str_1 \) and \( str_2 \) be two strings, then:

\[ str_1 \sqcup str_2 = \begin{cases} str & \text{if } str_1 = str_2 = str \\ \bot & \text{otherwise} \end{cases} \]

2.7.3 Special Features

There are several special features in FUF which are used to control the various aspects of FUF operation. These are cset, fset, pattern and control. Their usual place is in the grammar, but one can use them in input as well. I will say here a few words about them, a more detailed discussion will appear in 5.

- Fset is short for “Feature Set”.
  Is used to define the set of features appropriate for use under a particular feature. Figure 2.7 contains a fragment of a grammar, where the set of appropriate features for \texttt{grade} (in the given location) is \{1,2,3,4,5,6\}.

---

3In other words the unification succeeds if and only if \( r \leq s \), i.e. \( r \) is “under” \( s \) in the type hierarchy \( T \).

4Like the previous footnote, only \( r \prec s \), i.e. \( r \) is “strictly under” \( s \) in the type hierarchy \( T \).
Cset is short for “Constituent Set”. There are special parts in FDs called constituents. They play a crucial role in unification, serving as the entry points where the unification with the grammar starts. Initially the root of the FD is the first constituent, in the process of enriching the FD, other constituents are discovered and get unified in some particular order. Cset is used to explicitly identify constituents (see Figure 2.7).

Pattern.
The unification of the grammar and an FD produces another FD, which is supposed to describe a phrase in some human spoken language. To actually get the phrase one must know the order in which the constituents are to be linearized. Pattern is used to impose order on the constituents. It is also used in implicit identification of constituents (when there is no explicit cset available). (see Figure 2.7)

Control.
Allows to constrain feature values using the language of the underlying interpreter. Naturally, its syntax depends on the interpreter, taking Scheme, for example, a typical example of the usage of this feature is something like \( \text{control} \ (\text{>} \ 0 \ \text{number} \ 1) \). This particular statement succeeds if the feature \text{number} \ (at the same level as \text{control}) has a value greater than 1 (try to express the same thing without the control attribute). The control expression is evaluated at unification time.

Of course, all the above special features have special unification rules.
2.8 Existing Implementation

FUF was written in Common Lisp by Elhadad [3]. The most recent version is 5.3. In this implementation:

- The FD data structure is based on association lists (lists of attribute-value pairs)
- Paths are encoded as tagged lists of features
- Unification is based on continuations (fail/success) and undoable side effects (a stack of side effects is maintained)

2.9 Objectives of this Work

In this work, I have thrived to achieve three objectives:

1. Efficiency
   The goal is to implement FUF efficiently with speed being the keyword. To meet this goal I had to find the most suitable data structures and algorithms.

2. Compatibility
   The new implementation must be compatible with the already existing one. This means that given an FD and a grammar it should produce the same unification result as the current implementation whenever it is possible.

3. Usability
   It should be as convenient to use as the current implementation. This means that the new interpreter must support tracing, debugging, and be opened to an interpreted underlying language to allow fast experimentation. This last constraint is also necessary to enable the user to express complex constraints using the control attribute.

2.10 Summary

In this chapter, I have:
briefly characterized FUF (sections 2.1 through 2.5),

given short examples of an FD, grammar and their unification (2.6),

defined the important structures of FUF (2.7),

In the following chapter, I present the structure of the CFUF interpreter, starting with the internal encoding of FDs, the translation of FUF grammars, the unification process and the treatment of special features. Finally, I present the detailed evaluation of the implementation.
Chapter 3

FD Translation

3.1 Overview

In this chapter, I define the internal representation into which an FD specification is translated and describe the translation process itself. This process is called FD compilation. In effect, this process defines the semantics for FDs with the semantic domain being the set of all the finite rooted directed graphs, where:

- the arcs are labeled by feature names
- the nodes are typed (see 2.7.2)
- no two arcs from the same node carry the same label

There are two levels of reasoning about the FD graph. The first level (the low level) is in terms of the intermediary graph $G_{fd}$, which is a tree. The second level (the high level) is in terms of the quotient graph $\hat{G}_{fd}$, which is a finite rooted directed graph. Accordingly, I will describe FD compilation in this chapter as a two-stage process: parsing into the $G_{fd}$ structure; then construction of the complete internal structure $\hat{G}_{fd}$.

The parser drives the FD translator by issuing the “FD Compile Commands” (FDC commands). Should a command fail, the corresponding FD is reported as invalid. The FDC commands are shown on Table 3.1.

One can think of the whole translation process as a run of a virtual machine, where the FDC commands are the machine instructions and the quotient graph is its memory.
CHAPTER 3. FD TRANSLATION

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize()</td>
<td>Realize()</td>
</tr>
<tr>
<td>AddLeaf(feature, type)</td>
<td>AddLeaf(type)</td>
</tr>
<tr>
<td>AddPath(feature, path)</td>
<td>AddPath(path)</td>
</tr>
<tr>
<td>EnterSubgraph(feature)</td>
<td>EnterSubgraph(upcount)</td>
</tr>
<tr>
<td>LeaveSubgraph()</td>
<td>AddSpecial(feature, special type)</td>
</tr>
</tbody>
</table>

Table 3.1: FDC commands

3.2 Intermediary Graph

Let FD be a feature description, then the FD intermediary graph is $G_{fd} = \langle V, E, F, \sim \rangle$ where

- $F$ is the set of the features found in the FD specification.
- $V$ is the set of the graph nodes, called the individual nodes. There is a type $\text{type}(n)$ associated with each node $n \in V$.
- $E = \{ (n, \text{feat}, m) \in V \times F \times V \}$ is the set of the graph edges labeled by features.
- $\sim$ is an equivalence relation defined on the graph paths (see definition 1). This equivalence relation is induced by the FD.

Both during FD translation and FD–Grammar unification, the graph $G_{fd}$ undergoes changes. Those changes are the result of applying the FDC commands produced by the parser in the former case or stored in the Grammar in the latter.

Definition 1 Path.

A path is a sequence of features $\langle f_0, f_1, \ldots, f_{k-1} \rangle \in F^k$, such that the following holds:

$$\exists (m_0, m_1, \ldots, m_k) \in V^{k+1} (m_0 \text{ is the root of } G_{fd} \text{ and } \bigwedge_{i=0}^{k-1} (\langle m_i, f_i, m_{i+1} \rangle \in E))$$

---

1I will extend the notion of node type when I talk about backtracking in section 6.5.
CHAPTER 3. FD TRANSLATION

The node \( m_k \) is called the **destination node of the path** \( \langle f_0, f_1, \ldots, f_{k-1} \rangle \) and is denoted by \( \text{dst}((f_0, f_1, \ldots, f_{k-1})) \).

The path \( \langle f_0, f_1, \ldots, f_{k-1} \rangle \) is called the **location of the node** \( m_k \) and is denoted by \( \text{loc}(m_k) \) (\( \text{loc} = \text{dst}^{-1} \)).

\( P(G_{fd}) \) denotes the set of all the paths in the graph \( G_{fd} \).

**Definition 2** Relative path.
A FUF path \( \langle \text{up} g_0 g_1 \ldots g_{l-1} \rangle \) (or \( \{g_0 g_1 \ldots g_{l-1}\} \)) is represented by the pair \( \langle \text{up}, \langle g_0, g_1, \ldots, g_{l-1} \rangle \rangle \), called a **relative path**, where \( \text{up} \) is either a positive integer or \(-1\) and \( g_i \in F \):

\[
\begin{align*}
i(\{g_0 g_1 \ldots g_{l-1}\}) & = \langle -1, g_0, g_1, \ldots, g_{l-1} \rangle \\
i(\langle \text{up} g_0 g_1 \ldots g_{l-1} \rangle) & = \langle \text{up}, \langle g_0, g_1, \ldots, g_{l-1} \rangle \rangle
\end{align*}
\]

The only use for a relative path is as the second argument of the follow function or AddPath command:

\[
\text{follow}(\langle f_0, f_1, \ldots, f_{k-1} \rangle, \langle \text{up}, \langle g_0, g_1, \ldots, g_{l-1} \rangle \rangle) = \left\{ \begin{array}{ll}
\langle g_0, g_1, \ldots, g_{l-1} \rangle & \text{if} \ \text{up} = -1 \ \vee \ \text{up} \geq k \\
\langle f_0, f_1, \ldots, f_{k-1} \rangle & \text{otherwise}
\end{array} \right.
\]

This is because “relative path” is a syntactic notion – not a semantic one. The destination of a relative path depends on the “current location” when the relative path is met and interpreted during the traversal of an FD specification.

**Definition 3** Node children and descendants.

\[
\text{child}(n, \text{feat}) = \left\{ \begin{array}{ll}
m & \text{if} \ \langle n, \text{feat}, m \rangle \in E \ \land \ \text{type}(m) \neq \bot \\
\varepsilon & \text{otherwise}
\end{array} \right.
\]

The value \( \varepsilon \) means that the particular node does not have the particular child. I need it for the function \( \text{child} \) to be fully defined.

\( C(n) = \{ m \mid \exists f \in F \text{ such that } \text{child}(n, f) = m \in V \} \) — the set of the node’s children.

\( D(n) = \{ n \} \cup C(n) \cup (\cup_{m \in C(n)} D(m)) \) — the set of the node’s descendants.

Assuming \( C(n \in V) \neq \emptyset \), what is the type of the node \( n \)? It cannot be provided explicitly by the user, because ordinarily (attr type) appearing in the FD specification corresponds to a leaf node typed with type (the only exceptions are (attr nil) and (attr any)). To type the nodes like
n (called **internal nodes**), two new types (privately used by this FUF implementation) are introduced — in\textsubscript{nil} and in\textsubscript{any}. The in\textsubscript{nil} type is the usual type of internal nodes. An internal node typed with in\textsubscript{any} is guaranteed to be ground after unification succeeds, meaning that one of its descendants will have a type \( t \in T \). That is, an in\textsubscript{any} node leads to any leaf. Naturally, the types in\textsubscript{nil} and in\textsubscript{any} are called the **internal types**.

**Definition 4 Child type.**

type\textsubscript{child} : \( V \times F \rightarrow T_{\text{all}} \):

\[
\text{type}_{\text{child}}(n, f) = \begin{cases} 
\text{type}(m) & \text{if } \text{child}(n, f) = m \in V \\
\text{nil} & \text{else if } \text{type}(n) \notin \{ \text{none} \} \cup T \\
\text{none} & \text{otherwise}
\end{cases}
\]

**Definition 5 Type–Node consistency.**

There are two functions devoted to type–node consistency handling — \( \text{cons} : V \times T_{\text{all}} \rightarrow \{ \text{false}, \text{true} \} \) and \( \text{cons}_{\text{child}} : V \times F \times T_{\text{all}} \rightarrow \{ \text{false}, \text{true} \} \):

\[
\text{cons}(n, t) = \begin{cases} 
\text{true} & \text{if } \text{type}(n) \cup t \neq \bot \\
\text{false} & \text{else if } \text{type}(n) \notin \{ \text{in}_{\text{nil}}, \text{in}_{\text{any}} \} \\
\text{true} & \text{if } t = \text{given} \land \exists m \in D(n)(\text{type}(m) \in T) \\
\text{true} & \text{else if } t = \text{none} \land \forall m \in D(n)(\text{type}(m) \notin \{ \text{any} \} \cup T) \\
\text{true} & \text{else if } t \in T \land \forall m \in D(n)(\text{type}(m) \notin \{ \text{any}, \text{none} \} \cup T) \\
\text{false} & \text{otherwise}
\end{cases}
\]

\( \text{cons}_{\text{child}}(n, f, t) \) is defined identically to \( \text{cons}(n, t) \), except that \( \text{type}(n) \) is replaced with \( \text{type}_{\text{child}}(n, f) \).

A node \( n \in V \) and type \( t \in T_{\text{all}} \) are said to be consistent if and only if \( \text{cons}(n, t) = \text{true} \).

A node \( n \in V \), feature \( f \in F \) and type \( t \in T_{\text{all}} \) are said to be consistent if and only if \( \text{cons}_{\text{child}}(n, f, t) = \text{true} \).

A node \( n \in V \) can have a feature \( f \in F \) of type \( t \in T_{\text{all}} \) if and only if \( n, f \) and \( t \) are consistent.

Note that the \( \text{cons} \) predicate is sensitive to the order of its arguments: \( \text{cons}(\text{in}_{\text{nil}}, t \in T) \) may be \( \text{true} \), but \( \text{cons}(t \in T, \text{in}_{\text{nil}}) \) is always \( \text{false} \).
The respective FDC commands:
1  initialize
2    enter subgraph a
3      add leaf c 1
4    leave subgraph
5    enter subgraph b
6      add leaf d 2
7    leave subgraph
8    add path a {b}
9  realize

Figure 3.1: Merger example

Definition 6 Type update.
The functions \( \text{upd} : V \times T_{all} \rightarrow T_{all} \) and \( \text{upd}_{\text{child}} : V \times F \times T_{all} \rightarrow T_{all} \)
return the updated type for a node \( n \) and \( \text{child}(n, f) \) respectively.

\[
\text{upd}(n, t) = \begin{cases} 
\bot & \text{if cons}(n, t) = \text{false} \\
\text{type}(n) \sqcup t & \text{else if type}(n) \sqcup t \neq \bot \\
\text{type}(n) & \text{else if } t = \text{given} \\
t & \text{otherwise}
\end{cases}
\]

\( \text{upd}_{\text{child}}(n, f, t) \) is defined identically to \( \text{upd}(n, t) \), except that \( \text{type}(n) \) and \( \text{cons}(n, t) \) are replaced with \( \text{type}_{\text{child}}(n, f) \) and \( \text{cons}_{\text{child}}(n, f, t) \) respectively.

3.3 Quotient Graph

Consider the FD on Figure 3.1. When the time comes to execute command 8, the paths \( p_1 = \langle a \rangle \) and \( p_2 = \langle b \rangle \) are made equivalent. Note that making two paths equivalent may affect other paths as well, in our example \( p_1 \sim p_2 \) causes \( \langle a, c \rangle \sim \langle b, c \rangle \land \langle a, d \rangle \sim \langle b, d \rangle \).

Definition 7 Node equivalence.
Let FD be a functional description and \( G_{fd} = \langle V, E, F, \sim \rangle \) be the FD's
intermediary graph. Then I define a node equivalence relation $\sim_G$:

$$\forall n_1, n_2 \in V \ (n_1 \sim_G n_2 \iff \exists p_1, p_2 \in P(G_{fd}) \ (n_i = \text{dst}(p_i) \land p_1 \sim p_2))$$

Let $G_{fd}$ be the intermediary graph, then the quotient graph is $\tilde{G}_{fd} = \langle V_{\sim_G}, E_{\sim_G}, F \rangle$:

- $V_{\sim_G} = \{[n]_{\sim_G} | n \in V\}$ — the set of the graph nodes, called the composite nodes.

- $E_{\sim_G} = \{\langle N, \text{feat}, M \rangle \in V_{\sim_G} \times F \times V_{\sim_G} | \exists \langle n, m \rangle \in N \times M \text{ such that } \langle n, \text{feat}, m \rangle \in E\}$

The definitions of path, node location and path destination remain the same, except that instead of individual nodes and paths we deal with composite nodes and path equivalence sets. Thus $\text{dst}_{\sim_G}(\{p \in P(G_{fd})\}_G) = \{\text{dst}(p)\}_{\sim_G}$ and $\text{loc}_{\sim_G} = \text{dst}_{\sim_G}^{-1}$.

Given a node $N \in V_{\sim_G}$ and a relative path $\langle up, \{g_0, g_1, \ldots, g_{l-1}\} \rangle$ one can follow that path from the node $N$ using the function $\text{follow}$ defined in 3.2. A very important thing about relative paths is that the absolute paths $\text{follow}(p, \langle up, \{g_0, g_1, \ldots, g_{l-1}\} \rangle)$ and $\text{follow}(q, \langle up, \{g_0, g_1, \ldots, g_{l-1}\} \rangle)$ may not be equivalent even though $p \sim q$. Consider the FD on Figure 3.2. Although $\langle a, c \rangle \sim \langle b, c \rangle$, $\text{follow}(\langle a, c \rangle, \{1, \{d\}\}) = \langle a, d \rangle \not\sim \langle b, d \rangle = \text{follow}(\langle b, c \rangle, \{1, \{d\}\})$. This re-enforces the point made earlier that relative paths are a syntactic notation and cannot be maintained as such in the semantic domain.

Let us assume for the moment that there is a function $\text{type}_{\sim_G} : V_{\sim_G} \rightarrow T_{alt}$ which returns the type of a composite node (see definition 9 in the next section). Then we can define the $\text{child}_{\sim_G}$ function similarly to $\text{child}$:

**Definition 8 Composite node children and descendants.**

$$\text{child}_{\sim_G}(N, \text{feat}) = \begin{cases} M & \text{if } \langle N, \text{feat}, M \rangle \in E_{\sim_G} \land \text{type}_{\sim_G}(M) \neq \bot \\ \varepsilon & \text{otherwise} \end{cases}$$

$$C_{\sim_G}(N) = \{M | \exists \text{feat} \in F \text{ such that } \text{child}_{\sim_G}(N, \text{feat}) = M\}$$

$$D_{\sim_G}(N) = \{N\} \cup C_{\sim_G}(N) \cup (\bigcup_{M \in C_{\sim_G}(N)} D_{\sim_G}(M))$$

Of course, the similarity in definitions of the functions $\text{child}$ and $\text{child}_{\sim_G}$ is not incidental. Composite nodes are like individual nodes for all intent
The FDC commands:
initialize
enter subgraph a
add path c \{b, c\}
add leaf d 1
leave subgraph
enter subgraph b
add leaf d 1
leave subgraph
realize

Figure 3.2: \( \text{follow}((\langle a, c\rangle, \langle 1, \langle d\rangle \rangle) \neq \text{follow}((\langle b, c\rangle, \langle 1, \langle d\rangle \rangle)) \)

and purpose, both kinds of nodes support the same interface. This makes the usage of composite nodes completely transparent. And in the spirit of this transparency, each function for individual nodes has a respective counterpart in the domain of composite nodes:

- Child type: \( \text{type}_{\text{child} \sim G} : V_{\sim G} \times F \rightarrow T_{\text{all}} \).
- Type–Node consistency: \( \text{cons}_{\sim G} : V_{\sim G} \times T_{\text{all}} \rightarrow \{ \text{false, true} \} \) and \( \text{cons}_{\text{child} \sim G} : V_{\sim G} \times F \times T_{\text{all}} \rightarrow \{ \text{false, true} \} \).
- Type update: \( \text{upd}_{\sim G} : V_{\sim G} \times T_{\text{all}} \rightarrow T_{\text{all}} \) and \( \text{upd}_{\text{child} \sim G} : V_{\sim G} \times F \times T_{\text{all}} \rightarrow T_{\text{all}} \).

Their definitions and meanings are analogous to those for individual nodes.

From now on I will omit the \( \sim G \) subscript from the names of functions. It will be clear from the context (or inessential to know) which version is being used.

### 3.3.1 Node Merger Implementation

The process of making the paths \( p_1 \) and \( p_2 \) equivalent, called node merger, involves merging the composite nodes \( \text{dst}(p_1) \) and \( \text{dst}(p_2) \) in one. The cru-
cial thing about node merger is that one must be able to undo the merger relatively fast. Indeed, an FD is almost always doomed to participate in unification and to have different node merger techniques for compilation and unification is too troublesome (furthermore it is unreasonable, since $G_{fd}$ is modified by the same set of FDC commands both in compilation and unification). But efficient unification implementation is impossible without efficient implementation of backtracking, which essentially means that changes must be easily undoable.

I see two ways to accomplish node merger:

1. **Active merger.**
   
   All the properties of one of the two nodes are merged with those of the other by physically moving them. Let $N_2 \in V_{\sim G}$ be the "donor" and $N_1 \in V_{\sim G}$ be the "recipient", then the following steps are to be taken:
   
   - Disconnect all the children of $N_2$ and connect them to $N_1$.
   - Make all the references to $N_2$ point to $N_1$ instead. The obvious references are those of the parents of $N_2$, since $N_2$ is no longer their child, but $N_1$ is. Besides the parents, FUF special types (cset, pattern) may have references to FD nodes and those must be updated as well.
   - Merge the local data of $N_2$ with $N_1$.
   - Destroy the node $N_2$.

2. **Passive merger.**
   
   Instead of moving the properties, link the two nodes together.

Both ways have their pros and cons\(^2\). In order to choose between the two approaches, one has to evaluate their performance on the four major node operations: node merger, node merger undo, get node type, get node child. My findings show that the passive merger is best suited for the first two operations and it is as effective as the active merger for the latter two in most of the cases. For a detailed evaluation data see 7.3.2.

In order for the passive merger be efficient it must support:

---

\(^2\)Another way would be to combine the two. I did not explore such a hybrid approach, but I would like to in the future.
• fast find–representative operation.

• convenient iteration over the individual nodes comprising the compound node.

The former calls for the **Union–Find** organization of the individual nodes (as is typical for algorithms relying on the maintenance of an equivalence relation), while the latter — for a cyclic list. Naturally, I combine the two. Every individual node has a link consisting of two pointers — **ufnext** and **cnex**.

The ufnext pointer serves to find the representative (the ufnext pointee is called the **uf–neighbor**) and the cnex points the next individual node in the cycle (called the **c–neighbor**).

When two composite nodes are merged, the respective representatives are linked together and the representative of the new composite node is choosen among them. As can be seen, merging two nodes requires update of only four pointers — ufnext and cnex pointers of the first representative and the same for the second. Refer to Figure 3.3 for visual feedback on the node merger.

Let \(N \in V_{\sim G}\), then the functions \(repr : V_{\sim G} \rightarrow V\), \(cnex : V_{\sim G} \rightarrow V\) and \(ufnext : V_{\sim G} \rightarrow V\) return the representative, c–neighbor and uf–neighbor respectively.

**Definition 9 Composite node type.**

Let \(N \in V_{\sim G}\), then \(type_{\sim G}(N) = type(repr(N))\).

Node merger may be quite time consuming, regardless of the technique used to implement it. Let \(N, M \in V_{\sim G}\) be two composite nodes, then their merger may cause other nodes to merge. For example, if there exists a feature \(f \in F\), such that \(child(N, f) = N' \in V_{\sim G}\) and \(child(M, f) = M' \in V_{\sim G}\), then the nodes \(N'\) and \(M'\) have to be merged as well. Indeed, the merger of \(N\) and \(M\) results in a new node which replaces the two and inherits all their children. Let \(K\) denote that node, then \(\langle K, f, N'\rangle, \langle K, f, M'\rangle \in E_{\sim G}\) which is permitted only if \(N'\) and \(M'\) are the same.

**Definition 10** \(merge : V_{\sim G} \times V_{\sim G} \times 2^{V_{\sim G} \times V_{\sim G}} \rightarrow 2^{V_{\sim G} \times V_{\sim G}}\)

Let \(N, M \in V_{\sim G}\) be two composite nodes, then \(merge(N, M, \emptyset)\) returns the set of all the node pairs to be merged along with the two included. If the merger cannot be performed \(\emptyset\) is returned.
Figure 3.3: Node merger.

An algorithm to find $mergeset(N, M, S)$:

if $N = M \vee \langle N, M \rangle \in S$ return $S$
if $\text{type}(N), \text{type}(M) \in \{\text{in}_{nil}, \text{in}_{any}\}$ then
  foreach $\langle N', M' \rangle \in V^2_{\sim_G}$ such that
    $\exists f \in F \ (\text{child}(N, f) = N' \land \text{child}(M, f) = M')$ do
      $S := mergeset(N', M', S)$
      if $S = \emptyset$ return $\emptyset$
    end
  return $S \cup \{\langle N, M \rangle\}$
endif
if $\text{type}(M) \notin \{\text{in}_{nil}, \text{in}_{any}\}$ then
  if $\text{cons}(N, \text{type}(M))$ return $S \cup \{\langle N, M \rangle\}$
  return $\emptyset$
endif
if $\text{type}(N) \notin \{\text{in}_{nil}, \text{in}_{any}\} \land \text{cons}(M, \text{type}(N))$ return $S \cup \{\langle N, M \rangle\}$
return $\emptyset$
### Definition 11

MergeNodes\((N, M)\) denotes the operation of merging two composite nodes \(N, M \in V_{\sim\alpha}\).

An algorithm for MergeNodes\((N, M)\):

\[
S := \text{mergeset}(N, M, \emptyset) \\
\text{if } S \neq \emptyset \text{ then} \\
\quad \text{foreach } \langle N, M \rangle \in S \text{ do} \\
\quad \quad V_{\sim\alpha} := V_{\sim\alpha} \setminus \{N, M\} \cup \{N \cup M\} \\
\quad \quad n := \text{repr}(N) \\
\quad \quad m := \text{repr}(M) \\
\quad \quad k \in \{n, m\} \text{ is the representative of the new node } N \cup M \\
\quad \quad \text{ufnext}(n) := k \\
\quad \quad \text{ufnext}(m) := k \\
\quad \quad \begin{pmatrix} c\text{next}(m) \\ c\text{next}(n) \end{pmatrix} := \begin{pmatrix} c\text{next}(n) \\ c\text{next}(m) \end{pmatrix} \\
\quad \text{end} \\
\text{else} \\
\quad \text{the merger of } N \text{ and } M \text{ fails} \\
\text{endif}
\]

All the node operations can be roughly divided to operations aware of the representative and those which are not. The former class of operations needs the representative in order to complete and thus they follow the \textbf{ufnext} pointers, while the latter do not care what node is the representative and they follow the \textbf{cnext} pointers. Table 3.2 lists the operations.

<table>
<thead>
<tr>
<th>action</th>
<th>\textbf{ufnext} pointer</th>
<th>\textbf{cnext} pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>find/add node child</td>
<td>doesn’t care</td>
<td>read</td>
</tr>
<tr>
<td>get/set node type</td>
<td>read</td>
<td>doesn’t care</td>
</tr>
<tr>
<td>(undo) merge nodes</td>
<td>read/write</td>
<td>read/write</td>
</tr>
</tbody>
</table>

Table 3.2: Node operations
The respective FDC commands:
1 initialize
2 enter subgraph \{a b\}
3 add path c \{^d\}
4 leave subgraph
5 add leaf a 1
6 realize

Figure 3.4: Node/Arc removal

### 3.3.2 Node/Arc Validity

Sometimes during compilation or unification, the need to remove a node or an arc arises. Figure 3.4 shows a scenario where both arcs and nodes have to be removed. On the left of the figure there is an example FD along with the FDC commands issued by the parser. The leftmost graph is how \( \bar{G}_{fd} \) looks just before command 5 is executed. At that point the current node is the root. The rightmost graph is the \( \bar{G}_{fd} \) after the command 5 is done. Note that once the type of the node \( N_1 \) is set to 1, the nodes \( N_2 \) and \( N_3 \) and the corresponding arcs \( \langle N_1, b, N_2 \rangle, \langle N_2, c, N_3 \rangle \) and \( \langle N_2, d, N_3 \rangle \) have to be removed since in FUF only leaf nodes can have non predefined types. Physical removal of nodes and arcs brings forth too many troubles\(^3\). So neither arcs nor nodes are ever physically deleted. Instead they are invalidated. A node is invalid if its type is \( \perp \) and an arc is invalid if the node it points to is invalid.

Another point to ponder, is that for every graph \( \bar{G}_{fd} \) there exists an fd FDO, such that \( \bar{G}_{fd} = \bar{G}_{fd_0} \) and during the construction of which no deletion of nodes/arc ever takes place. The graph on Figure 3.4 can be produced by

---

\(^3\)One source of troubles are the special types, which may have reference to nodes. What can be more disastrous than a dangling reference? Also undo of node removal is expensive.
FD specification:
```
((a ((b 1)
    (c {^ ^ d})
    ({} {e}))))
```

FD parse tree

Figure 3.5: FD translation scheme.

- (((a 1)) with the latter being true.

### 3.4 The Translation Process

For the time being I will ignore the special features since they are not part of the FD per se. They act more like directives to FUF controlling various aspects of compilation, unification and linearization.

Figure 3.5 schematically depicts the translation process. Of course, an efficient implementation would skip the “Parse Tree Traversor” stage — the “Parser” can be linked to “FDVM” directly (this how I do it, actually).

#### 3.4.1 FD Parse Tree Traversal

The FD parse tree is a tree, with the arcs annotated by <attr> and leaf nodes — by <type> or <path> (see 2.5). Figure 3.6 shows an FD, along with the corresponding parse tree.
CHAPTER 3. FD TRANSLATION

The “Parse Tree Traversor” module issues the FDC commands as it traverses the parse tree depth-first, left-to-right.

Let $PT_{fd}$ be the parse tree produced for the FD. The following functions are used in describing the traversal procedure:

- $\text{root}(PT_{fd})$ — the root node of the parse tree
- $C_{PT}(x)$ — the set of the children of the node $x$
- $\text{attr}(x)$ — the annotation on the arc connecting the node $x$ to its parent
- $\text{value}(x)$ — the annotation of the node $x$, either type or path (defined only if $C_{PT}(x) = \emptyset$)
- $\text{is\_path}(a)$ — true if the annotation $a$ is a path

A procedure for the traversal of the parse tree $PT_{fd}$:

```plaintext
procedure TraverseParseTree(PTfd)
    issue(Initialize())
    if $C_{PT}(\text{root}(PT_{fd})) = \emptyset$ then
        issue(AddLeaf(\text{value}(\text{root}(PT_{fd}))))
    else
        TraverseNode(\text{root}(PT_{fd}))
    endif
    issue(Realize())

procedure TraverseChild(feat, x)
    if $C_{PT}(x) \neq \emptyset$ then
        issue(EnterSubgraph(feat))
        TraverseNode(x)
        issue(LeaveSubgraph(\))
    else if (is\_path(\text{value}(x))) then
        issue(AddPath(feat, \text{value}(x)))
    else
        issue(AddLeaf(feat, \text{value}(x)))
    endif

procedure TraverseChild(x)
    if $C_{PT}(x) \neq \emptyset$ then
        TraverseNode(x)
    else if (is\_path(\text{value}(x))) then
        issue(AddPath(\text{value}(x)))
    endif
```

procedure TraverseNode(cur)

```plaintext
foreach x ∈ $C_{PT}(\text{cur})$ do
    d := 0
    feat\_present := true
    if is\_path(\text{attr}(x)) then
        let attr(x) = \langle \text{up}, \langle g_0, g_1, \ldots, g_{l-1} \rangle \rangle
        issue(EnterSubgraph(\text{up}))
        while d < l - 1 do
            d := d + 1
            if feat\_present then
                feat := g_{l-1}
            endif
            else
                feat := \text{attr}(x)
            endif
            issue(EnterSubgraph(g_d))
        end
    endif
endif
```
else
  \( \text{issue}(\text{AddLeaf}(\text{value}(x))) \)
endif

\( \text{issue}(\text{LeaveSubgraph}()) \)

\( d := d - 1 \)
\end

Figure 3.7 shows the traversal of the parse tree from Figure 3.6, where:

<table>
<thead>
<tr>
<th>FDC Command Template</th>
<th>( \alpha = \text{EnterSubgraph}(a) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_1 = \text{EnterSubgraph}(a) )</td>
<td>( \alpha_2 = \text{AddLeaf}(b,1) )</td>
</tr>
<tr>
<td>( \alpha_2 = \text{AddLeaf}(b,1) )</td>
<td>( \alpha_3 = \text{AddPath}(c,\langle 2, \langle d \rangle \rangle) )</td>
</tr>
<tr>
<td>( \alpha_3 = \text{AddPath}(c,\langle 2, \langle d \rangle \rangle) )</td>
<td>( \alpha_4 = \text{EnterSubgraph}(-1) )</td>
</tr>
<tr>
<td>( \alpha_4 = \text{EnterSubgraph}(-1) )</td>
<td>( \alpha_5 = \text{AddPath}(\langle -1, \langle e \rangle \rangle) )</td>
</tr>
<tr>
<td>( \alpha_5 = \text{AddPath}(\langle -1, \langle e \rangle \rangle) )</td>
<td>( \alpha_6 = \alpha_7 = \text{LeaveSubgraph}() )</td>
</tr>
<tr>
<td>( \alpha_6 = \alpha_7 = \text{LeaveSubgraph}() )</td>
<td>( \text{EnterSubgraph}(\text{feat}) )</td>
</tr>
<tr>
<td>( \text{EnterSubgraph}(\text{feat}) )</td>
<td>( \text{AddLeaf}(\text{feat}, \text{type}) )</td>
</tr>
<tr>
<td>( \text{AddLeaf}(\text{feat}, \text{type}) )</td>
<td>( \text{AddPath}(\text{feat}, \text{path}) )</td>
</tr>
<tr>
<td>( \text{AddPath}(\text{feat}, \text{path}) )</td>
<td>( \text{EnterSubgraph}(\text{upcount}) )</td>
</tr>
<tr>
<td>( \text{EnterSubgraph}(\text{upcount}) )</td>
<td>( \text{AddPath}(\text{path}) )</td>
</tr>
<tr>
<td>( \text{AddPath}(\text{path}) )</td>
<td>( \text{LeaveSubgraph}() )</td>
</tr>
<tr>
<td>( \text{LeaveSubgraph}() )</td>
<td>( \text{LeaveSubgraph}() )</td>
</tr>
</tbody>
</table>
And Figure 3.8 shows the stages of the quotient graph, from prior $\alpha_1$ to after $\alpha_7$.

The procedure is a pretty straightforward DFS traversal of a tree. The only thing worth of extra attention is the handling of path attributes. To keep the FDC commands as simple as possible (but no simpler), there are no EnterSubgraph(path), AddLeaf(path,type) or AddPath(path,path) commands, though attribute–value pairs like ($\{^\wedge f g\} (\{h t\})$), ($\{^\wedge f g\} t$) or ($\{^\wedge 2\} \{^\wedge f g\}$) are legitimate in an FD specification. These “missing” commands are expressed in terms of the existing ones, using the fact that ($\{^\wedge up g_0 g_1 \ldots g_{l-1} \text{ value}\}$) has the same meaning as ($\{^\wedge up\} ((g_0\ldots((g_{l-1} \text{ value})\ldots)))$).

Footnote: The FD syntax provides for no way to simplify ($\{^\wedge up\} \text{ value}$) ($l = 0$). That is where the featureless forms of AddLeaf/AddPath are used.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Precondition</th>
<th>Semantics</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(CN)</td>
<td></td>
<td>returns the number of elements</td>
<td></td>
</tr>
<tr>
<td>val(CN)</td>
<td>size(CN) &gt; 0</td>
<td>returns the topmost element</td>
<td></td>
</tr>
<tr>
<td>push(_{CN}(N))</td>
<td></td>
<td>Saves the node (N) on top of the stack.</td>
<td>size(CN) is incremented by one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Always returns true.</td>
<td>val(CN) = N</td>
</tr>
<tr>
<td>pop(_{CN}())</td>
<td>size(CN) &gt; 0</td>
<td>Removes the topmost element off the stack.</td>
<td>size(CN) is decremented by one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Always returns true.</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: CN register

3.4.2 FD Virtual Machine (FDVM)

The process of the FD translation can be viewed, as the execution of a virtual machine, the FDVM, whose memory is the quotient graph \(G_{fd}\) (section 3.3), instructions are the FDC commands (Figure 3.1) and which has 3 registers:

- PC — the current command (aka program counter)
- CN — the node \(N \in V_{\sim_G}\) to which the command pointed by PC is being applied (the current node)
- CP — the path, which was used to reach the current node (the current path)

The registers CN and CP are actually stacks of nodes and paths\(^5\) respectively. Table 3.3 describes the semantics of the CN register, (the semantics of the CP register is analagous).

**Definition 12 Internal nodes.**

A node \(N\) is called internal if and only if \(\text{type}(N) \in \{\text{in}_{ni}, \text{in}_{any}\}\). The predicate \(\text{internal} : V_{\sim_G} \rightarrow \{\text{false, true}\}\) is used to determine whether a node is internal.

Each FDC command has a precondition and a postcondition associated with it. The precondition must be true before the command application in order for it to succeed and the postcondition holds after the successful

---

\(^5\)In reality CP is a pair \((p, U)\), where \(p\) is the current path and \(U\) is a stack of undo information per each EnterSubgraph call. It is an implementation detail.
<table>
<thead>
<tr>
<th>Command</th>
<th>Precondition</th>
<th>Semantics</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitFD()</td>
<td>( \hat{G}_{fd} = (0, 0, F) \land \lceil CN \rceil = 0 \land \lceil CP \rceil = 0 )</td>
<td>Initializes the FD translation by creating the root node.</td>
<td>( G_{fd} = ({ N_0 }, \emptyset, F) \land \text{type}(N_0) = \text{in}_{\text{init}} \land \lceil CN \rceil = 0 \land \lceil CP \rceil = 0 \land \text{val}(CN) = () )</td>
</tr>
<tr>
<td>Realize()</td>
<td>( \lceil CN \rceil = 1 \land \text{val}(CN) = N_0 \land \lceil CP \rceil = 0 )</td>
<td>Signals the end of FD translation.</td>
<td>( \text{type}(CN) \neq \text{none} )</td>
</tr>
<tr>
<td>AddLeaf(t)</td>
<td>( \lceil CN \rceil &gt; 0 \land \text{type}(CN) = \text{none} \land \lceil CP \rceil = 0 )</td>
<td>Add/s update a child of the current node.</td>
<td>( \text{type}(CN) = \text{none} )</td>
</tr>
<tr>
<td></td>
<td>( \text{cons}(CN, f, t) \land \text{cons}(CN, f, \text{in}_{\text{init}}) )</td>
<td></td>
<td>( \text{type}(CN) = \text{none} )</td>
</tr>
<tr>
<td>AddLeaf(t)</td>
<td>( \lceil CN \rceil &gt; 0 \land \text{cons}(CN, t) )</td>
<td>Updates the current node.</td>
<td>( \text{type}(CN) = \text{none} )</td>
</tr>
<tr>
<td></td>
<td>( \text{type}(CN) = \text{none} \land \text{cons}(CN, f, \text{in}_{\text{init}}) )</td>
<td></td>
<td>( \text{type}(CN) = \text{none} )</td>
</tr>
<tr>
<td>EnterSubgraph(L)</td>
<td>( \lceil CP \rceil &gt; 0 \land \text{follow}(CP, (\text{wp}, (l))) = p \land \text{dist}(p) = M \in V_{\text{in}} \land \text{type}(M) = \text{none} )</td>
<td>Makes a child of the current node.</td>
<td>( \text{type}(CN) = \text{none} )</td>
</tr>
<tr>
<td></td>
<td>( \text{cons}(CN, f, t) \land \text{cons}(CN, f, \text{in}_{\text{init}}) )</td>
<td></td>
<td>( \text{type}(CN) = \text{none} )</td>
</tr>
<tr>
<td>EnterSubgraph(up)</td>
<td>( \lceil CP \rceil &gt; 0 \land \text{follow}(CP, (\text{wp}, (l))) = p \land \text{dist}(p) = M \in V_{\text{in}} \land \text{type}(M) = \text{none} )</td>
<td>Makes a node on the current path.</td>
<td>( \text{push}<em>{CP}(p) \land \text{push}</em>{CN}(M) )</td>
</tr>
<tr>
<td>LeaveSubgraph()</td>
<td>( \lceil CN \rceil &gt; 0 \land \lceil CP \rceil &gt; 0 )</td>
<td>Restores the previous current node and path.</td>
<td>( \text{pop}<em>{CN}() \land \text{pop}</em>{CP}() )</td>
</tr>
</tbody>
</table>

Table 3.4: FDC commands (all, but AddPath)

application. The types \( \text{in}_{\text{init}} \), \( \text{in}_{\text{any}} \) and \( \bot \) can never be arguments of any FDC command, as they are not part of the FUF syntax. This is an implicit preconditions for the AddLeaf command.

Table 3.4 lists all the FDC commands, but the AddPath, which is easy to describe in terms of the AddLeaf, EnterSubgraph/LeaveSubgraph FDC commands and the MergeNodes operation.

### 3.5 Summary

This section describes the translation of an FD into the internal representation. I first define the internal representation (3.2 and 3.3), then the translation process itself is described, introducing the notion of FD Virtual Machine (3.4).

The internal representation consists of two layers — the intermediary

---

\(^6\)For the sake of convenience I use simply \( CN \) and \( |CN| \) instead of \( \text{val}(CN) \) and \( \text{size}(CN) \).
graph (3.2), which is always a directed tree, and the quotient graph (3.3), which is a rooted directed graph.

An FD is translated to the respective quotient graph by running the FDVM, which has:

- memory — the quotient graph
- instruction set — the FDC commands
- registers — CP (current path), CN (current node) and PC (program counter — current FDC command)
Chapter 4

Grammar Translation

4.1 Overview

Grammars, like FDs, must be translated to some internal representation before they can be used in unification. Although grammars look very similar to FDs (in fact, any FD can be used as a grammar having no choice points) the respective internal representations are very different. This is because they have very distinct roles — grammars act as programs and FDs as inputs.

The grammar internal representation is an AND–OR tree (also called the grammar tree), where all the nodes are divided to and–nodes and or–nodes corresponding to the grammar alts and branches respectively. The or–nodes are the grammar choice points and the and–nodes contain the FDC commands of the respective grammar branches. These FDC commands are executed by the FDVM (see 3.4.2) during the unification.

The grammar translation process could have been presented in a fashion similar to FD translation — by introducing some Grammar Virtual Machine and defining its operation. It is, however, an overkill, as the grammar compilation is a simple process, compared to that of FD.

4.2 Grammar Tree

Internally, a grammar is represented by a tree $T_{gr}$. In a manner similar to FD compilation, grammar compilation is driven by the GC commands issued by the parser. While almost all the FDC commands can actually modify the
CHAPTER 4. GRAMMAR TRANSLATION

graph \( \bar{G}_{fd} \) during FD compilation, those commands have no effect on the structure of \( T_{gr} \). This is because, as an FD is parsed, the FDC commands are immediately executed. In contrast a grammar is kept as a set of FDC commands, which we only execute during unification. \( T_{gr} \) is shaped by the other four commands of the GC command set.

The formal definition of \( T_{gr} = \langle V, E \rangle \) follows.

- The nodes of \( T_{gr} \) fall into two categories: \( V = V_{and} \cup V_{or} \) (\( V_{and} \cap V_{or} = \emptyset \)). Let \( \langle x, y \rangle \in V_{and} \times V_{or} \), then \( x \) is an and-node and \( y \) is an or-node. Another names for \( x \) and \( y \) are a branch-node and an alt-node respectively.

- \( E = \{\langle x, y \rangle\} \) - the edges of the tree \( T_{gr} \). Let \( x \in V \) be a tree node, then \( C_T(x) \) denotes the set of its children: \( C_T(x) = \{y \mid \langle x, y \rangle \in E\} \). An important property of the tree \( T_{gr} \) is that an and-node can be connected only to an or-node and vice versa: \( \forall x \in V_{and}(C_T(x) \subseteq V_{or}) \land \forall y \in V_{or}(C_T(y) \subseteq V_{and}) \).

- The root of \( T_{gr} \) is a branch-node \( x_0 \in V_{and} \).

- All the leaves of the tree \( T_{gr} \) are branch-nodes.

As the names suggest alt-nodes correspond to the alts of the Grammar and branch-nodes — to its branches. Consider the expression ((alt \( branch_1 \) \( branch_2 \ldots branch_n \))) which is valid in Grammars only. Table 4.1 shows the GC commands issued by the parser when it is seen.

This particular sequence of GC commands, when executed, creates an alt-node \( y \) and \( n \) branch-nodes \( x_1, x_2, \ldots, x_n \), such that \( C_T(y) = \{x_i\}_{i=1}^{n} \), with the branch-node \( x_i \) corresponding to the \( i \)'s branch in the alt \( y \). If, for example, \( branch_1 \) has the layout of ((alt (...) \( alt (...) \) \ldots \( alt (...) \))) — the total of \( m \) alts, then there are \( m \) alt-nodes \( y_1, y_2, \ldots, y_m \) such that \( C_T(x_1) = \{y_j\}_{j=1}^{m} \), with the alt-node \( y_j \) being the \( j \)'s alt in the branch \( x_1 \) (see Figure 4.1).

But what about the FDC commands participating in Grammar compilation? Table 4.2 shows a general sequence of commands delimited by an EnterBranch/LeaveBranch pair. The union of the chunks 1 through \( m \)

\(^1\)With one exception — adding a pattern changes the tree \( T_{gr} \). See section 5.4.1 for the details.
enter alt
  enter branch
    [GC commands for \textit{branch}_1]
  leave branch
  enter branch
    [GC commands for \textit{branch}_2]
  leave branch
  \vdots
  enter branch
    [GC commands for \textit{branch}_n]
  leave branch
leave alt
enter branch
  FDC commands chunk 1
  enter alt
  ...
  leave alt
  FDC commands chunk 2
  enter alt
  ...
  leave alt
  :
  FDC commands chunk \( m - 1 \)
  enter alt
  ...
  leave alt
  FDC commands chunk \( m \)
leave branch

| Table 4.2: \(((alt (\ldots)) (alt (\ldots)) \ldots (alt (\ldots)))\) |

This gives us the FDC commands for which the branch is responsible, if during unification this branch is chosen then these commands are immediately applied to the current node of \( G_{fd} \) and if one of them fails — the branch fails. Of course, there can be (and will be) more FDC commands inside the EnterAlt/LeaveAlt blocks, but other branches are responsible for them.

Let \( x \in V_{\text{and}} \) be a Grammar branch, then \( \text{cmds}(x) \) denotes the commands for which the branch \( x \) is responsible. For the sake of convenience I define \( \text{alts}(x) = C_T(x) \) and \( \text{branches}(y) = C_T(y) \) where \( \langle x, y \rangle \in V_{\text{and}} \times V_{\text{or}} \).

At last, but not least, consider a trivial grammar on Figure 4.2. The Figure stresses an essential detail of the Grammar compilation. Despite the fact that the two chunks of FDC commands for which the root branch \( x_0 \) is responsible (namely [AddLeaf a 1, EnterSubgraph b] and [LeaveSubgraph, AddLeaf e 4]) are separated by the alt, they are going to be executed consecutively! The implication of the above is that when it is \( x_1 \)'s turn to execute its FDC commands ([AddLeaf c 2]) the current node of \( G_{fd} \) is not the node for which \( \text{cmds}(x_1) \) are intended (assuming the current constituent is the root) — \( \text{cmds}(x_1) \) are supposed to be applied to \( \text{dst}(\langle b \rangle) \), but the current node
\[ CN = dst(\langle \rangle) = \text{root}(G_{fd}). \]

Thus the notion of alt location is introduced — it is a path \( p \in P(G_{fd}) \), such that for every branch \( x \) belonging to the alt in question, \( cmds(x) \) are to be applied to \( dst(p) \) (relative to the current constituent). For \( y \in V_{\text{and}} \), \( \text{loc}(y) \) denotes the location of the alt \( y \).

Let \( x \in V_{\text{and}} \), then:

\[
\text{loc}(x) = \begin{cases} 
\text{loc}(y) & \text{if } y \in V_{or} \land x \in \text{branches}(y) \\
\langle \rangle & \text{if } x \text{ is the root branch}
\end{cases}
\]

### 4.3 Summary

This chapter describes the internal representation of a FUF grammar. The notions of grammar tree and alt location are introduced. The main intuition is that a grammar is compiled into a set of FDC instructions organized as an AND–OR tree. The CFUF interpreter will traverse this tree and execute the appropriate chunk of FDC commands in the right context (as indicated by the alt–location annotations) on the current FD.

We have ignored in this chapter the handling of FUF’s special features (fset, cset, pattern, control). These require a special treatment, because their
semantics is different from the graph–based semantics implemented by the set of FDC commands. They are described in the next chapter. Eventually, the main task of the CFUF interpreter will be to handle the non–deterministic traversal of the and–or tree in an efficient manner. This is the topic of the following chapter.
Chapter 5

Special features

5.1 Overview

FUF has special features which govern various aspects of FD construction and unification. For example, any grammar with any self respect, will contain the pattern special feature to specify the linearization order. See 2.7.3 for a brief overview of the existing special features. In this chapter, I define the precise rules according to which the special features fset, cset and pattern are interpreted: which values they can accept and how they are unified.

Definition 13 Special node. Let $N, M \in V_{\sim_{\ell}}$ and special $\in \{fset, cset, pattern\}$, such that $child(N, special) = M$. Then the node $M$ is called an fset node, a cset node or a pattern node for special being fset, cset or pattern respectively.

Note that there is no control node. This is not an error, the control special feature does not affect $\bar{G}_{fd}$ in any way. It is used to specify arbitrary queries about the current state of the quotient graph $\bar{G}_{fd}$ using the underlying interpreter language. The appropriate FDC command AddSpecial(control,query) fails if the query fails.

5.2 Fset

Definition 14 Appropriate features. Let $N \in V_{\sim_{\ell}}$ be a node of the graph $\bar{G}_{fd}$, then appropriate($N$) $\subseteq F$ denotes
the set of all the non special features the node \( N \) is allowed to have:

\[
f \notin \text{appropriate}(N) \land \text{child}(N, f) = M \in V_{\sim_G} \longrightarrow \text{type}(M) = \text{none}
\]

Such features are called \textbf{features appropriate for} \( N \).

The \texttt{fset} attribute is used to control the set of appropriate features for \( N \in V_{\sim_G} \).

If \( \text{child}(N, \text{fset}) = \varepsilon \), then \( \text{appropriate}(N) = F \). If, however, \( \text{child}(N, \text{fset}) = M \in V_{\sim_G} \), then \( \text{appropriate}(N) = \text{type}(M) = \{f_i\}_{i=0}^{k-1} \), \( \text{appropriate}(N) = \emptyset \) for \( k = 0 \).

Let \( M_1, M_2 \in V_{\sim_G} \) be two fset nodes with \( \text{type}(M_1) = \text{fset}_1 \) and \( \text{type}(M_2) = \text{fset}_2 \), then \( \text{type}(M_1) \sqcap \text{type}(M_2) = \text{fset}_1 \cap \text{fset}_2 \).

Note that this definition of the fset unification is in sharp contrast with the usual definition as used in other typed feature structure (TFS) unification formalisms (cf. [1], [13], [2]). In these formalisms, all nodes have an appropriateness restriction defined. When two nodes are unified, the fset equivalent of the unifier is the union of the fset of the nodes (and not the intersection as is the case here). This dual interpretation of the appropriateness restriction is related to the role that we give to partial information. By default, in FUF, nodes do not carry appropriateness restrictions. In other words, the default value of the fset of a node is the (possibly infinite) set of attributes \( F \). In other TFS formalisms, there is no way to express infinite appropriate sets.

The practical implications are minor, but the implementation effect is tremendous: in all TFS formalisms where a finite fset is implied, the internal representation for feature structures can be the implied fixed-arity term where all appropriate attributes appear. In this case, the same optimizations applicable to Prolog first-order terms (position-based indexing of the children arcs) can be used. In FUF, in contrast, this approach is not practical, as the potential number of outgoing arcs can grow to unbounded sizes.

### 5.3 Cset

Cset is used in FUF to identify special nodes within an FD which are called the \textit{constituents}. Recall that FUF is designed to support a functional analysis of linguistic information, where sub-FDs are labeled by the function of the
sub–FD within the containing FD. A functional analysis, however, implies that an underlying structural analysis be performed. For example, in the functional analysis of the clause John eats an apple given as example in the introduction, the description of John as actor implies that the John be identified as a structural constituent within the clause. This is in contrast to the word an, which taken in isolation is not a constituent of the clause.

In addition, in the FD shown in Fig.5.1, there is an essential distinction between the feature tense and the other three features. The difference is that tense encodes a property of the clause as a whole, whereas the three other features describe the sub–constituents of the clause. Nothing in the syntax of the FD distinguishes between these two roles (attribute and subconstituent). In fact, this is one of the major attractions of the FUF formalism, because it allows the uniform encoding of different constituent structures within the same FD. This is a critical property which allows FUF to encode pipelines of grammars dealing with orthogonal structures in the same description (lexical structure, semantic structure, syntactic structure).

The functional treatment of the FD, however, requires the distinction of property and sub-constituent. The cset feature is introduced in FUF to allow the expression of this distinction.

Figure 5.2 indicates how the cset feature is used. Cset lists the features within an FD which are to be considered sub–constituents. Note that the value of cset is a list of individual nodes within the FD. The formal definition of the cset unification rule is the following:

Let \( N \in V_{\sim g} \) such that \( \text{child}(N, \text{cset}) = M \in V_{\sim g} \), then \( \text{type}(M) = \langle A, B, I, D \rangle \in V^4 \) (not \( V_{\sim g}^4 \)). The cset type has references to the individual nodes and not composite. In fact, any special type having references to nodes, ought to reference individual nodes only. This is because composite
Figure 5.2: Using cset to distinguish sub-constituents from other features

nodes come and go, but individual nodes always remain\(^1\). The four cset components stand for Absolute, Base, Incremental and Decremental.

The cset special type is used to indentify subconstituents of the node \(N\) (provided the node is a constituent itself) during the unification.

Let \(M_1, M_2 \in V_\text{cset}\) be two cset nodes with \(\text{type}(M_1) = \langle A_1, B_1, I_1, D_1 \rangle\) and \(\text{type}(M_2) = \langle A_2, B_2, I_2, D_2 \rangle\), then:

\[
\text{type}(M_1) \sqcup \text{type}(M_2) = \begin{cases} 
\langle \emptyset, B_1 \cup B_2, I_1 \cup I_2, D_1 \cup D_2 \rangle & \text{if } A_1 = A_2 = \emptyset \\
\langle A_1, \emptyset, \emptyset, \emptyset \rangle & \text{if } A_1 \neq \emptyset \land A_2 = \emptyset \land \psi(A_1) \\
\langle A_2, \emptyset, \emptyset, \emptyset \rangle & \text{if } A_2 \neq \emptyset \land A_1 = \emptyset \land \psi(A_2) \\
\langle A, \emptyset, \emptyset, \emptyset \rangle & \text{if } A_1 = A_2 = A \neq \emptyset \land \psi(A) \\
\bot & \text{otherwise}
\end{cases}
\]

\[\psi(A) \iff (I_1 \cup I_2) \subseteq A \land (D_1 \cup D_2) \cap A = \emptyset\]

5.4 Pattern

While cset is used to describe the constituent structure of an FD, the pattern special feature is used to describe the linear ordering of the constituents within the linguistic structure.

Figure 5.3 illustrates how the pattern special feature is used. In the example, the order of apparition of the sub-constituents actor, action and object is specified by the pattern feature.

\(^1\)This argument is a pure formality. The implementation always operates with individual nodes.
<table>
<thead>
<tr>
<th>action</th>
<th>eat</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
<td>John</td>
</tr>
<tr>
<td>object</td>
<td>apple</td>
</tr>
<tr>
<td>tense</td>
<td>present</td>
</tr>
<tr>
<td>cset</td>
<td>(action actor object)</td>
</tr>
<tr>
<td>pattern</td>
<td>(actor action object)</td>
</tr>
</tbody>
</table>

Figure 5.3: Specifying the order of sub-constituents with pattern

Formally, patterns are ordered vectors of normal and mergeable constituents and the special symbol $\delta$ denoting *dots* (see 2.5). Mergeable constituents are a special feature of the FUF language [3].

Let $N \in V_{\sim_G}$, such that $\text{child}(N, \text{pattern}) = M \in V_{\sim_G}$, then $\text{type}(M) = \langle c_0, c_1, \ldots, c_{k-1} \rangle$, where $c_i \in \{\delta\} \cup V \cup \{\hat{n} \mid n \in V\}$.

**Definition 15 Pattern constituents.**

Let $p = \langle c_0, c_1, \ldots, c_{k-1} \rangle$ be a pattern, then $\text{val}(p) = \{n \in V \mid n = c_i \lor \hat{n} = c_i \text{ for some } 0 \leq i \leq k-1\}$ — is the set of all the constituents referenced by the pattern $p$.

An ordinary constituent is denoted by an individual node $n \in V$ (as a pattern specification contains references to nodes, just like cset). A mergeable constituent — by $\hat{n}$, where $n \in V$ is an individual node. For example, let the current path register (see 3.4.2) $CP = \langle f_0, f_1, \ldots, f_{k-1} \rangle$, then the pattern specification ($\text{dots a (* a) dots (* \{b c\})}$) is represented by the vector $\langle \delta, n, \hat{n}, \delta, \hat{m} \rangle$, where $n = \text{dst}(\langle f_0, f_1, \ldots, f_{k-1}, a \rangle)$ and $m = \text{dst}(\langle b, c \rangle)$.

Patterns impose order on the constituents which is preserved throughout the unification of patterns.

Pattern unification may yield more than one result provided that each one obeys the order constraints of all the patterns participating in the unification. Indeed, consider the patterns $\langle \delta, n_1, \delta, m_1, \delta \rangle$ and $\langle \delta, n_2, \delta, m_2, \delta \rangle$,.
their unification produces 6 different results:

\[
\langle \delta, n_1, \delta, m_1, \delta \rangle \sqcup \langle \delta, n_2, \delta, m_2, \delta \rangle = \left\{ \begin{array}{l}
\langle \delta, n_1, \delta, m_1, \delta, n_2, \delta, m_2, \delta \rangle \\
\langle \delta, n_1, \delta, n_2, \delta, m_1, \delta, m_2, \delta \rangle \\
\langle \delta, n_1, \delta, n_2, \delta, m_2, \delta, m_1, \delta \rangle \\
\langle \delta, n_2, \delta, n_1, \delta, m_2, \delta, m_1, \delta \rangle \\
\langle \delta, n_2, \delta, \delta, n_1, \delta, m_1, \delta \rangle \\
\langle \delta, n_2, \delta, \delta, m_2, \delta, m_1, \delta \rangle 
\end{array} \right. 
\]

Note that for each result, \( n_1 \) precedes \( m_1 \) and \( n_2 \) precedes \( m_2 \) — the order is preserved.

**Definition 16 Source/destination patterns.**

Let \( p \) be a pattern participating in pattern unification. Then \( p \) is called a source pattern.

Let \( q \) be one possible outcome of pattern unification, then \( q \) is called a destination pattern.

Unification of patterns is in effect string matching, where:

1. \( \delta \) matches zero or more pattern elements
2. a constituent does not match anything, but itself
3. a mergeable constituent matches any constituent, which can be merged with it by the MergeNodes operation (see page 29)
4. a destination pattern cannot have repetitions of the same constituent, except if and only if this constituents recurs in a source pattern (the repetition constraint)

Examples:

1. \( \langle \delta \rangle \sqcup \langle n, \delta, m, k \rangle = \langle n, \delta, m, k \rangle \)
2. \( \langle n, \delta \rangle \sqcup \langle \delta, m \rangle = \langle n, \delta, m \rangle \)
3. \( \langle n \rangle \sqcup \langle \hat{m} \rangle = \langle \hat{n} \rangle = \langle \hat{m} \rangle \) — the nodes \([n] \sim_G\) and \([m] \sim_G\) are merged
4. \( \langle n, \delta \rangle \sqcup \langle \delta, n \rangle = \langle n \rangle \) — because neither of the source patterns has two occurrences of \( n \), \( \langle n, \delta, n \rangle \) is not a valid destination pattern
In the following discussion I ignore the mergeable constituents and the repetition constraint. The implementation takes them into account, but the presentation becomes too complex when they are introduced.

The rest of the section presents a way to find the destination patterns. It is not how it actually works in CFUF, as I cannot ignore mergeable constituents or the repetition constraint there, but this way it can be visualized easily.

**Definition 17 Pattern prefix/core/suffix.**
Let $p = \langle c_0, c_1, \ldots, c_{k-1} \rangle$ be a pattern. Then $\text{prefix}(p)$, $\text{suffix}(p)$ and $\text{core}(p)$ return the prefix, suffix and core of the pattern $p$ respectively:

\[
\begin{align*}
\text{prefix}(p) &= \begin{cases} 
\langle c_0, c_1, \ldots, c_{l-1} \rangle & \text{if } c_i \neq \delta \text{ for } 0 \leq i \leq l - 1 \text{ and } c_l = \delta \\
\emptyset & \text{otherwise (} l = k \text{ — the pattern does not have dots)} 
\end{cases} \\
\text{suffix}(p) &= \begin{cases} 
\langle c_l, c_{l+1}, \ldots, c_{k-1} \rangle & \text{if } c_i \neq \delta \text{ for } l \leq i \leq k - 1 \text{ and } c_{k-1} = \delta \\
\emptyset & \text{otherwise (} l = 0 \text{ — the pattern does not have dots)} 
\end{cases} \\
\text{core}(p) &= \begin{cases} 
\langle c_l, c_{l+1}, \ldots, c_{m-1} \rangle & \text{if } \left( \text{prefix}(p) = \langle c_0, c_1, \ldots, c_{l-1} \rangle, \text{suffix}(p) = \langle c_m, c_{m+1}, \ldots, c_{k-1} \rangle \right) \\
\emptyset & \text{otherwise (} \text{prefix}(p) = \text{suffix}(p) = \emptyset \text{)}
\end{cases}
\]

If a pattern $p$ has a non empty core, then this core starts and ends with dots.

**Definition 18 Prefix/suffix consistency.**
The patterns $p_1$ and $p_2$ are said to be prefix-consistent if and only if there exists $i \in \{1, 2\}$, such that $\text{prefix}(p_i)$ is a prefix of $p_{3-i}$. The predicate $\text{cons}_{\text{prefix}}$ is used to denote the prefix-consistency — $\text{cons}_{\text{prefix}}(\{p_1, p_2\}) = \text{true} \iff$ the patterns $p_1$ and $p_2$ are prefix-consistent.

The suffix-consistency is defined analogously, except that every instance of the word prefix is replaced by suffix.

For example:

- the patterns $\langle n_1, m_1, \delta, k_1 \rangle$ and $\langle n_1, \delta, m_2 \rangle$, $\langle n_1 \rangle$ and $\langle n_1, m_2 \rangle$ are prefix-consistent, while $\langle n_1, m_i \rangle$ and $\langle n_1, m_2 \rangle$ are not.
the patterns \( \langle n_1, \delta \rangle \) and \( \langle n_2, m_2, k_2 \rangle \), \( \langle n_1 \rangle \) and \( \langle m_2, \delta, n_1 \rangle \) are suffix-consistent, while \( \langle m_1, n_1 \rangle \) and \( \langle m_2, n_1 \rangle \) are not.

The notion of prefix-consistency can be extended to the set of patterns \( P = \{ p_i \}_{i=1}^n \):

\[
\text{cons}_{\text{prefix}}(P) = \begin{cases} 
\text{true} & \text{if } n < 2 \\
\land_{p_i, p_j \in P} \text{cons}_{\text{prefix}}(\{p_i, p_j\}) & \text{otherwise}
\end{cases}
\]

In other words, the set of patterns is prefix-consistent if and only if every two patterns from the set are prefix-consistent. Suffix-consistency is extended exactly the same way.

Let \( P = \{ p_i \}_{i=1}^n \) be the source patterns, then \( q \) is a destination pattern if and only if the following holds:

1. \( \text{cons}_{\text{prefix}}(P) \land \text{cons}_{\text{suffix}}(P) \)
2. \( \text{prefix}(q) = \max_{p_i \in P} (\text{prefix}(p_i)) \land \text{suffix}(q) = \max_{p_i \in P} (\text{suffix}(p_i)) \)
3. \( \text{val}(q) = \cup_{i=1}^n \text{val}(p_i) \)
4. Let \( p_i \in P \), such that \( n, m \in \text{val}(p_i) \) and \( p_i = \langle \ldots n \ldots m \ldots \rangle \) (in other words \( n \) remains before \( m \)).
5. Let \( p_i \in P \), such that \( n, m \in \text{val}(p_i) \) and \( p_i = \langle \ldots n \ldots m \ldots \rangle \) (in other words \( n \) remains adjacent to \( m \)).
6. Let \( p_i \in P \), and for all \( x \in \cup_{i=1}^n \text{val}(p_i) \), let \( \text{rep}(x, p_i) \) be the number of repetitions of \( x \) in \( p_i \), then: \( \text{rep}(x, q) = \max_i (\text{rep}(x, p_i)) \). That is, unification of patterns does not remove nor add repetitions.

The last condition enforces that \( \langle \delta, n, \delta, m \rangle \) and \( \langle \delta, n, \delta \rangle \) do not produce \( \langle \delta, n, n, \delta, m \rangle \), because the number of repetitions would change from 1 to 2 for \( n \).

It follows from condition 2, that all the destination patterns share the same prefix and suffix, called the destination prefix and destination suffix.

Let \( \text{dstpat}(P) \) denote the set of all the destination patterns of the set \( P \), then \( P \) is said to be consistent if and only if \( \text{dstpat}(P) \neq \emptyset \).
Let \( P = \{p_i\}_{i=1}^n \) be a set of \( n \) patterns. Let \( p_i \in P \) be such that \( \text{prefix}(p_i) = p_i \) (that is, \( p_i \) is dotless). In this case if \( \text{dstpat}(P) \neq \emptyset \) then \( \text{dstpat}(P) = \{p_i\} \). The reader can check, that this is a direct consequence of the conditions 1–6.

Let \( P = \{p_i\}_{i=1}^n \) be a set of patterns, such that there are no dotless patterns. In addition, \( \text{cons}_{\text{prefix}}(P) \land \text{cons}_{\text{suffix}}(P) \). I am going to show that \( P \) is consistent, by finding the set \( \text{dstpat}(P) \neq \emptyset \).

If the condition 1 holds (which is the case), then we can find the destination prefix and suffix. Once we have found them we can assume w.l.o.g. that \( p_i = \text{core}(p_i) \) for all \( p_i \in P \).

Let \( P = \{p_i\}_{i=1}^n \) be a set of patterns, such that \( \text{core}(p) = p \) (i.e., all the patterns begin and end with \( \text{dots} \)). A pattern \( p \in P \) can be viewed as a 1–dimensional grid consisting of directed arcs (constituents) connected by knots (\( \text{dots} \)). Refer to Figure 5.4 for examples.

The set \( P \) can be viewed as an \( n \)-dimensional grid, composed of the pattern grids (see Figure 5.5). There are two special knots — the source and the destination. No arc ever enters the source and no arc ever leaves the destination (like the graph flow).

Figure 5.6 shows a possible grid for the 3 patterns \( \langle \delta, n_i, \delta, m_i, \delta \rangle \), where \( i = 1, 2, 3 \). The marked path corresponds to a destination pattern \( \langle \delta, n_2, \delta, m_1, \delta, n_1, \delta, m_3, \delta \rangle \).

It is easy to see, that the set of all the paths from the source to the destination induces the set of all the destination patterns — the conditions 1 and 2 are given and the conditions 3–5 follow from the way the grid is built.

To summarize the above, to find the \( \text{dstpat}(P) \), the following steps are to be taken:

1. Check \( P \) for dotless patterns. If found — make sure that \( P \) is consistent.
(dots l dots) + (dots n m dots k dots)

Figure 5.5: Set of patterns as grid

Figure 5.6: A possible 3-dimensional grid for \( \langle \delta, n_i, \delta, m_i, \delta \rangle \)
CHAPTER 5. SPECIAL FEATURES

If so, then the dotless pattern is the only destination path.

2. \( P \) has no dotless patterns. If \( cons_{\text{prefix}}(P) \land cons_{\text{suffix}}(P) \) — acquire the destination prefix and suffix and proceed further.

3. Build the grid of pattern cores.

4. Any path from the grid source knot to the destination one induces a destination core. Prepending the destination prefix and appending the destination suffix yields a destination pattern.

5.4.1 Patterns and Grammar

Pattern unification produces multiple results, which are to be tried one after the other until either FD-Grammar unification completes successfully or fails. Therefore, pattern unification creates a choice point, which must be reflected in the grammar tree \( T_{gr} \). It is, however, prohibitively expensive to modify the structure of the tree \( T_{gr} \) dynamically during unification. This means, that not all the choice points are identified. While some may call it a bug, I prefer to call it a feature (which is, by the way, consistent with the original LispFUF implementation).

The command \( AddSpecial(pattern, p) \) is the only FDC command, which affects the \( T_{gr} \) during the grammar compilation. It creates a new alt-node with the only child branch-node, called pat-alt-node and pat-branch-node respectively. During the unification this pat-branch-node iterates over the set of pattern unification results, i.e. applying the \( \text{right_sibling} \) function to it produces the next result (or fails if no more). It is important to understand, that only those patterns unified as the result of AddSpecial command execution have choice points present in \( T_{gr} \) and thus get retried. Those, which get unified as the result of AddPath command execution are not retried, since doing so requires dynamic modification of the grammar tree structure.

For example, the command 3 in the following code forces the two patterns to unify, thus producing 6 different outcomes.

1: \( ((1 ((\text{pattern} \ (\text{dots} \ a \ \text{dots} \ b \ \text{dots})))))) \)
2: \( (2 ((\text{pattern} \ (\text{dots} \ c \ \text{dots} \ d \ \text{dots})))) \)
3: \( (1 \ {2})) \)
CHAPTER 5. SPECIAL FEATURES

Only the first of these six results, however, is considered. If the unification fails further on and retreats back to this point, it will not try another result. This is because no choice point corresponds to it. The whole branch fails, regardless of the fact that another destination pattern might fix the problem.

In this simple example, it is intuitively clear that no order selection could affect the end result of the unification (success or failure) since the order of the pattern could not possibly explain the failure or success.

However, matters get worse if the source patterns are mergeable (i.e., one of them has mergeable constituents). Recall, that unification of mergeable patterns may require unification of FD graph nodes (the so called side-effect of mergeable patterns unification). If the side-effect is undone another destination pattern must be picked. Pattern unification has not failed yet — it just “reviews other options”, contrary to the side-effect, which has failed. In other words the command which caused the patterns to unify in the first place (AddPath), and the side-effects occur in the different uticks (cf. Chapter 6). But different uticks correspond to different grammar branches. However, the AddPath command does not create a choice point!

Consequently, side-effects are executed only if there is a choice point corresponding to the pattern unification.

However, the mergeable patterns are rare and the speed gain is considerable.

Indeed, in the absence of side-effects the destination patterns differ only in the order they impose on the constituents. In most of the cases, the order does not matter. Thus retrying different destination patterns does not change the outcome, but only slows down the execution.

5.5 Summary

This chapter describes the three special features defined in FUF — fset, cset and pattern. The notion of appropriate features is defined in 5.2. The definition of constituents and their distinction from simple features is discussed in 5.3. The interaction of the pattern special feature and the grammar is discussed in 5.4.1.

For each of the special features, a special unification procedure has been defined which extends the semantics of FUF terms. In addition, for the cset and pattern features, special care must be paid to the fact that the values of
these features contain lists of references to other nodes within the FD.

In the past three chapters, I have described the internal encoding of FDs, grammars and their special features. We are now in a position to describe the unification process.
Chapter 6

Unification

6.1 Overview

The unifier works on compiled FDs and grammars. Intuitively, the task of the interpreter is to traverse the grammar AND-OR tree and to execute the chunks of FDC commands found in each and-node of the grammar on the current FD graph. In this chapter, we will focus on the handling of the non-determinism implied by the or-nodes. The execution of FDC commands in the right context is exactly similar to the original construction of the initial FD driven by the parser (recall from Chapter 3 that the parser emits FDC commands to construct the initial FD graph).

The major design decision in this chapter is to optimize the operation of backtracking. This is motivated by the empirical observation that FUF programs are extremely non-deterministic (e.g., they backtrack a lot). We will evaluate the well-foundedness of this working hypothesis in the next chapter.

The following code schematically depicts the process of FD-Grammar unification.

```
ok := true
ucount := 0
while ucount < ulimit do
  1: select a constituent ; if failed break
      while ucount < ulimit do
  2: select a Grammar branch ; if failed break
```


3: 
\( ucount := ucount + 1 \)

unify the current constituent with the current branch

ok := did unification succeed?

if not ok then

4: 
undo the changes made to \( \tilde{G}_f \)

endif

definition 19 Unification state.

- The unification process is said to be in the proceed state if the current branch has successfully unified with the current constituent (ok is true).

- The unification process is said to be in the retreat state if the current branch has failed to unify with the current constituent (ok is false).

The FD–Grammar unification fails if we either run out of constituents while in retreat state or the count of visited branches exceeds the given limit\(^1\). The unification succeeds otherwise.

As can be seen, FD–Grammar unification is quite a complex process. One can identify four essential tasks:

1. The current grammar branch selection.
2. The current constituent selection.
4. Undoing of the changes made by constituent–branch unification or simply backtracking.

Let us examine them more closely in the following sections.

\(^1\)It should be chosen reasonably to allow consistent FD and Grammar to unify, but to fail infinite or impractically time consuming unifications.
6.2 The Current Grammar Branch Selection

Three $T_{gr}$ traversal functions are used when determining the next branch to be visited:

- $parent: V_{and} \rightarrow V_{or} \cup \{\varepsilon\}$ or $parent: V_{or} \rightarrow V_{and}$.
  
  $parent(x \in V_{and}) = \begin{cases} 
  y \in V_{or} & \text{if } x \in \text{branches}(y) \\
  \varepsilon & \text{otherwise (x is the root branch)} 
  \end{cases}$
  
  $parent(y \in V_{or}) = x \in V_{and} \iff y \in \text{alts}(x)$
  
  For example, on Figure 4.1 $parent(x_i) = y$ and $parent(y_i) = x_1$.

- $first\_child: V_{and} \rightarrow V_{or} \cup \{\varepsilon\}$ or $first\_child: V_{or} \rightarrow V_{and}$.
  
  For $x \in V_{and}$ $first\_child(x)$ returns the first child of the branch-node $x$, which is an alt-node if exists at all, or $\varepsilon$ if $x$ is a leaf.
  
  For $y \in V_{or}$ $first\_child(y)$ returns the first child of the alt-node $y$, which is a branch-node.
  
  For example, on Figure 4.1 $first\_child(y) = x_1$ and $first\_child(x_1) = y_1$.

- $right\_sibling: V_{and} \rightarrow V_{and} \cup \{\varepsilon\}$ or $right\_sibling: V_{or} \rightarrow V_{or} \cup \{\varepsilon\}$.
  
  For $x \in V_{and}$ $right\_sibling(x)$ returns the right brother of the branch-node $x$, which is either another branch-node or $\varepsilon$ if $x$ is the last child of its parent.
  
  For alt-nodes $right\_sibling$ acts the similar way as for branch-nodes.
  
  For example, on Figure 4.1 $right\_sibling(x_1) = x_2$, $right\_sibling(x_n) = \varepsilon$, $right\_sibling(y_1) = y_2$ and $right\_sibling(y_m) = \varepsilon$.

The very first branch selected is the top level branch of the Grammar — the root node of $T_{gr}$. The way the next Grammar branch is selected depends on the unification state. If it is the proceed state — a new alt is entered. If it is the retreat state, we return to the previously visited alt “to give it another chance”. Let $Stack$ denote the stack of visited Grammar branches so far, the current branch is NOT on the stack. Below is the psuedocode for selecting the next branch:

**In the proceed state:**

- $push(Stack, current\_branch)$
- $alt := first\_child(current\_branch)$
if alt ≠ ε then
    make_current(first_child(alt))
    return true
endif
alt := parent(current_branch)
while alt ≠ ε
    next_alt := right_sibling(alt)
    if next_alt ≠ ε then
        make_current(first_child(next_alt))
        return true
    endif
    alt := parent(parent(alt))
end
return false

In the retreat state:
next_branch := current_branch
while not empty(Stack)
    next_branch := right_sibling(next_branch)
    if next_branch = ε then
        next_branch := pop(Stack)
        undo the changes made by the unification of next_branch with \( \bar{G}_{fd} \)
    else
        make_current(next_branch)
        return true
    endif
end
return false

Note, that each time we return to the previous alt (by doing pop(Stack)), we must undo the changes made to \( \bar{G}_{fd} \) by that alt.

6.3 The Current Constituent Selection

First, what is a constituent? As discussed in Sec.5.3, a constituent is a node in the \( \bar{G}_{fd} \), which has a special meaning to the Grammar (and the Grammar writer, of course). One can see a constituent as a subphrase in a larger phrase. The root of \( \bar{G}_{fd} \) is the very first constituent (it corresponds to the whole sentence). After it successfully unifies with the Grammar, its subconstituents are identified and unified with the Grammar recursively each one in turn and so on. In order for a node to be called constituent, it must
be selected by the constituent selection procedure, which is essentially driven by the cset features appearing in $\hat{G}_{id}$.

**Definition 20** Done constituent.
A constituent is called a **done constituent** if it has successfully unified with the Grammar.

**Definition 21** Standby constituent.
A constituent is called a **standby constituent** if it is not done and not current.

Done constituents are stored in $DoneStack$ stack, the most recent constituent being on the top. Standby constituents are stored in $StandbyQueue$ queue, the most recently selected constituents being at the rear. Thus a constituent is either done, standby or current.

The selection procedure depends on the unification state — whether it is proceed or retreat. The following steps are taken to determine the new current constituent:

**In the proceed state:**
- Identify the subconstituents (there may be none, see Figure 6.1) of the current constituent and enqueue them at the rear of $StandbyQueue$.
- Push the current constituent onto $DoneStack$, making it a done constituent.
- If $StandbyQueue$ is empty — fail, continue otherwise.
- Dequeue a standby constituent from the front of $StandbyQueue$ and make it the current one.

**In the retreat state:**
- If $DoneStack$ is empty — fail, continue otherwise.
- Enqueue the current constituent at the front of $StandbyQueue$, making it a standby constituent again.
- Pop the topmost done constituent off $DoneStack$ and make it the current one.
- Dequeue the standby constituents added most recently. They may no longer be the correct subconstituents of the current constituent.
S := \emptyset
M := \text{child}(N, \text{set})
\text{if } M \neq \emptyset \text{ then}
\quad \text{let } \text{type}(M) = \langle A, B, I, D \rangle
\quad \text{if } A \neq \emptyset \text{ return } \text{strip}(A)
\quad S := B \cup I
\quad \text{if } B \neq \emptyset \text{ return } \text{strip}(S \setminus D)
\text{end if}
S := S \cup \{ M \in C_{\sim \alpha}(N) \mid \text{type,child}(M,\text{cat}) \in T \cup \{ \text{any} \} \}
K := \text{child}(N, \text{pattern})
\text{if } K \neq \emptyset \text{ then}
\quad S := S \cup \text{val}(\text{type}(K))
\text{endif}
\text{return } \text{strip}(S \setminus D)
\text{Where } \text{strip}(X) = \{ L \in X \mid \text{type}(L) \in \{ \text{in, \textit{in} any} \} \text{ and } L \text{ is not already a constituent} \}.

Figure 6.1: The procedure to identify the subconstituents of the current constituent $N$

<table>
<thead>
<tr>
<th>Done</th>
<th>Current</th>
<th>Stand by</th>
</tr>
</thead>
<tbody>
<tr>
<td>proceed</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2 3</td>
</tr>
<tr>
<td>retreat</td>
<td>0 1</td>
<td>2</td>
</tr>
<tr>
<td>proceed</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>proceed</td>
<td>0 1</td>
<td>2</td>
</tr>
<tr>
<td>proceed</td>
<td>0 1 2</td>
<td>3</td>
</tr>
<tr>
<td>proceed</td>
<td>0 1 2 3</td>
<td>6</td>
</tr>
<tr>
<td>proceed</td>
<td>0 1 2 3 6</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: The scheme of constituent selection
Figure 6.3: Unification example with the given type

Figure 6.2 graphically demonstrates the constituent selection process. Now that we have identified the subconstituents in which order should we enqueue? This is not an idle question, but rather an important one, since it defines the order in which those subconstituents will be unified later with the Grammar. The point is, that not only can the order influence the speed of unification significantly, but it also can affect the outcome of the unification. Indeed, pure unification yields the same result no matter the order in which it proceeds, however, in the real world things are complicated by the presence of the given type. Figure 6.3 illustrates the matter.

The nodes \( N_1 = dst(a) \) and \( N_2 = dst(b) \) are the subconstituents of the root node \( N \) and the order in which they are visited is a matter of success and failure for the unification. The unification succeeds if \( N_1 \) precedes \( N_2 \) with the Figure 6.3 depicting the process and it fails otherwise. Indeed, after the root constituent is done, the unification cannot proceed with the constituent \( N_2 \) before \( N_1 \), because \( type_{ch\alpha}(N_2, feat) = nil \) and the type semantics states, that \( nil \perp given = \perp \).
CHAPTER 6. UNIFICATION

There is, however, another semantics\(^2\) which lets \textit{given} unify with \(type_{\text{child}}(N, \text{feat}) \in T\) only if the node \(N\) has already had the feature \textit{feat} prior the unification start.

The choice of semantics bears profound consequences. The first is easier to implement, it gives a user an ability to determine which constituent will be visited before which. The order of constituents, however, can affect the result of the unification, which is not so for the second semantics.

It appears that a good ordering can speed up the unification and a bad one can slow it down and even fail. My experience with the SURGE

English grammar [4, 5] shows that the order of the constituents as specified by the grammar in \texttt{cset} and \texttt{pattern} attributes sometimes performs very poorly. Instead of relying on that order (I call it \textbf{the default order}) I always sort the found subconstituents. The constituent \(N_1\) goes before \(N_2\) if the following is true:

\[
\varphi \leftarrow type_{\text{child}}(N_1, \text{cat}) \in T \land \\
\psi \leftarrow type_{\text{child}}(N_2, \text{cat}) \in T \land \\
\varphi \land \neg \psi \lor \varphi \leftarrow \psi \land N_1 \text{ is older than } N_2
\]

The rationale behind the above criterion is quite simple. I prefer to do first those constituents, which have the \texttt{cat} feature instantiated. This usually means that this constituent has a definite role and its unification is supposed to be more deterministic. Next if both constituents have (or do not have) the \texttt{cat} feature I choose the older among the two, i.e. the one which was added earlier during the unification and, as such, is likely to be richer (have more features) than the younger one. I stress the point that this ordering is based on observations and it does not guarantee to find the optimal ordering. However, it does perform better than the default one (see evaluation in 7).

\section{6.4 Constituent–Branch Unification}

Let:

- \(x \in V_{\text{and}}\) be the current Grammar branch.
- \(\text{loc}(x) = \langle g_0, g_1, \ldots, g_{m-1} \rangle\) be the location of the branch \(x\).

\(^2\)The current Lisp implementation does not support it, the new one supports it optionally and partially.
Chapter 6. Unification

- \( N \) be the current constituent.
- \( p = \langle f_0, f_1, \ldots, f_{k-1} \rangle \in \text{loc}(N) \) be the path, which was used to reach the node \( N \).

The unification is done by running the FDVM on the commands stored in the branch \( x \), when initially:

- \( PC \) is positioned at the first command
- \( CN = N \)
- \( CP = \langle f_0, f_1, \ldots, f_{k-1}, g_0, g_1, \ldots, g_{m-1} \rangle \)

The FDVM executes the commands precisely the same way, as during FD translation (see Table 3.4). The unification fails if a command fails.

6.5 Backtracking

The essence of backtracking is to undo the modifications made by the application of FDC commands. There are only three kinds of \( \mathcal{G}_{fd} \) modifications — arc addition, type change, node merger — and all of them must be undone. The arc addition is actually a private case of type change. Indeed, let \( N \in V_{\sim_0} \), such that \( \text{child}(N, \text{feat}) = \varepsilon \). Adding the arc \( \text{feat} \) yields \( \text{child}(N, \text{feat}) = M \neq \varepsilon \). In order to undo the addition of the new arc \( \text{feat} \) it suffices to change the type of \( M \) to \( \perp \). This invalidates both the node and the arc (see section 3.3.2), thus making \( \text{child}(N, \text{feat}) = \varepsilon \) again.

Two notions are crucial to understand the implementation of backtracking. These are unification clock (uclock) and unification tick (utick).

**Definition 22** Utick is an integer number associated with a Grammar branch. Each time a Grammar branch is made current a new utick is allocated and attached to the branch. The utick of the current branch is called the current utick.

**Definition 23** Uclock is a strictly increasing sequence of uticks. It acts both as a stack and a hash of uticks. As a stack it supports the operations push, pop, empty and as hash — find. A utick is said to be valid if uclock contains it. The current utick is returned by top(uclock).
Uclock is implemented as a bitmap and its capacity (denoted by \( \text{cap}_{\text{uclock}} \)) determines the maximum value for a utick. Uticks lie in the range \([-1, \ldots \text{cap}_{\text{uclock}} - 2]\). The utick \(-1\) corresponds to the FD translation stage and the initial value of the uclock is \((-1)\).

Each time a Grammar branch is made current (no matter in which state — proceed or retreat) a new utick is generated and pushed onto uclock, becoming the current utick\(^3\). Each time we return to the previous branch (in the retreat state) the topmost utick is popped from the uclock. In other words uclock accumulates the uticks of all the successful branches throughout the unification from the very start in the respective order — if \( i \) is a valid utick, then the Grammar branch with the associated utick \( i \) has successfully unified with the FD and has never been retreated. The opposite is also true.

To summarize, a utick is invalid if and only if the respective branch has failed.

In the following sections, I refine the concept of node type and node link (consisting of the ufnnext and cnext pointers, see section 3.3.1) with regard to the uclock and utick notions.

In the following discussion, I reason in terms of the individual nodes of the graph \( G_{fd} \) — the so called low level. The high level abstraction remains the same (no wonder, it is based on the low level).

### 6.5.1 Node type

Instead of a single value from \( T_{all} \) a node \( n \in V \) now contains a sequence \( \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{k-1}} \rangle \), where \( i_j \) are uticks and \( \tau_{i_j} \in T_{all} \setminus \{\bot\} \) for \( j = 0, 1, \ldots, k-1 \). Such a sequence is denoted by \( \text{type}_{\text{seq}}(n) \) and it obeys the following invariant:

- The sequence \( \langle i_0, i_1, \ldots, i_{k-1} \rangle \) is strictly increasing.

- There exists an integer \( 0 \leq m \leq k \), such that \( i_j \in \text{uclock} \iff j \in \{0, 1, \ldots, m-1\} \). In other words the sequence \( \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{k-1}} \rangle \) can always be divided into prefix \( \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{m-1}} \rangle \) and suffix \( \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_m}, \tau_{i_{m+1}}, \ldots, \tau_{i_{k-1}} \rangle \), such that only the prefix contains valid uticks. Such prefix is called the **valid prefix** (denoted by \( v\text{prefix}(\text{type}_{\text{seq}}(n)) \)) and the suffix — the

\(^3\)This means the \( \text{ulimit} \) variable (see the pseudocode in the overview, section 6) is always less than \( \text{cap}_{\text{uclock}} \)
valid suffix (denoted by \( vsuffix(type\text{-}seq(n)) \)). It is possible that the valid prefix or/and suffix be empty.

\[
\tau_{i_0} \supseteq \tau_{i_1} \supseteq \cdots \supseteq \tau_{i_{k-2}} \land (\tau_{i_{k-1}} \neq \text{none} \implies \tau_{i_{k-2}} \supseteq \tau_{i_{k-1}})
\]

The revised definition of \( type(n) \):

\[
\begin{align*}
\text{type}(n) &= \begin{cases} 
\tau_{i_{m-1}} & \text{if the valid prefix is } \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{m-1}} \rangle \\
\bot & \text{otherwise (} m = 0 \text{ --- the valid prefix is empty)}
\end{cases}
\end{align*}
\]

How is \( type\text{-}seq(n) \) modified? Suppose \( type\text{-}seq(n) = \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{k-1}} \rangle \), \( \langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{m-1}} \rangle \) is the valid prefix and the new type of the node \( n \) should be \( type \). First, the type \( type \) is either the result of unifying \( type(n) \) with some other type or \( none \). If the former is the case, then \( type(n) \supseteq type \). The latter happens if the node \( n \) is cancelled in the current branch --- \( type\text{-}seq(n) \) is not flushed, as doing so will make it impossible to undo the cancellation in the event of failure and subsequent backtracking.\(^4\) Second, \( type\text{-}seq(n) \) is updated as follows:

\[
\begin{align*}
type\text{-}seq'(n) &= \begin{cases} 
\langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{m-2}}, \tau_{i_{m-1}} = type \rangle & \text{if } top(uclock) = i_{m-1} \\\n\langle \tau_{i_0}, \tau_{i_1}, \ldots, \tau_{i_{m-1}}, \tau_{top(uclock)} = type \rangle & \text{if } top(uclock) \neq i_{m-1}
\end{cases}
\end{align*}
\]

A few notes:

- If \( top(uclock) \neq i_{m-1} \), then \( top(uclock) > i_{m-1} \). This is because both \( i_{m-1} \) and \( top(uclock) \) are valid uticks, but \( i_{m-1} \) precedes \( top(uclock) \) in the uclock, which is a strictly increasing sequence of uticks.

- If \( m = 0 \), then \( type\text{-}seq'(n) = \langle \tau_{top(uclock)} \rangle \), where \( \tau_{top(uclock)} = type \).

\(^4\)Though its type is not \( \bot \), but it is as good as invalid any way, since nothing can unify with \( none \), but \( none \) itself and \( nil \). Either way the respective feature becomes an “outlaw”.

\(^5\)Here one can optimize by ORing the test \( \tau_{i_{m-1}} = type \). In this case \( \tau_{i_{m-1}} = \tau'_{i_{m-1}} \) there is no need to duplicate \( \tau_{i_{m-1}} \) even though \( i_{m-1} \) is not the current utick. With this optimization the last utick of \( type\text{-}seq'(n) \) is not the current utick. This is actually the way I do it.
• Not only is \( \text{typeseq}(n) \) updated when a node type is changed, but it also
might be updated when a node type is just examined, since doing so
involves discarding the valid suffix. So two subsequent calls to \( \text{type}(n) \)
may not consume the same amount of time, as the second call does not
have to deal with the valid suffix — it is gone.

A very important consequence of the above is that \( \text{typeseq}(n) \) is ac-
tually annihilated when we want to know the type of a node, which
has been created during some failed branch unification. In this case
\( \text{typeseq}(n) = \text{vsuffix}(\text{typeseq}(n)) \).

The valid suffix of \( \text{typeseq}(n) \) disappears and the type updates performed
during the unification of the respective branches (which have failed, otherwise
they would not have been in the valid suffix) disappear along with it. This
is what we call backtracking.

6.5.2 Node Link

Let \( N, M \in V_{\alpha_{c}} \) and \( n = \text{repr}(N) \), \( m = \text{repr}(M) \), then merging the nodes
\( N \) and \( M \) is a matter of changing the node links of \( n \) and \( m \). To undo the
latest merger the latest node links must be restored. This is achieved the
same way node types are undone — through maintaining a sequence of links
\( \langle \lambda_{i_{0}}, \lambda_{i_{1}}, \ldots, \lambda_{i_{m-1}} \rangle \) denoted by \( \text{linkseq}(n) \), where \( \lambda_{i_{j}} = \langle \text{cnext}_{i_{j}}, \text{ufnext}_{i_{j}} \rangle \).
This sequence obeys invariant similar to that of \( \text{typeseq}(n) \), except that
the third item is not relevant here. The valid prefix and valid suffix
definitions remain unchanged and denoted with \( \text{vprefix}(\text{linkseq}(n)) \) and
\( \text{vsuffix}(\text{linkseq}(n)) \) respectively.

The revised definition of \( \text{cnext}(n) \) and \( \text{ufnext}(n) \):

\[
\langle \text{cnext}(n), \text{ufnext}(n) \rangle = \begin{cases} 
\lambda_{i_{m-1}} & \text{if the valid prefix is } \langle \lambda_{i_{0}}, \lambda_{i_{1}}, \ldots, \lambda_{i_{m-1}} \rangle \\
\langle n, n \rangle & \text{otherwise (} m = 0 \text{ — the valid prefix is empty) }
\end{cases}
\]

Suppose the node \( N \) with \( n = \text{repr}(N) \) and \( \text{linkseq}(n) = \langle \lambda_{i_{0}}, \lambda_{i_{1}}, \ldots, \lambda_{i_{m-1}} \rangle \)
is merged with another node and as the result of the merger we expect
\( \langle \text{cnext}(n), \text{ufnext}(n) \rangle \) be equal \( \text{link} \). Then the new value of \( \text{linkseq}(n) \) is:

\[
\text{linkseq}'(n) = \begin{cases} 
\langle \lambda_{i_{0}}, \lambda_{i_{1}}, \ldots, \lambda_{i_{m-2}}, \lambda'_{i_{m-1}} = \text{link} \rangle & \text{if } \text{top(uclock)} = i_{m-1} \\
\langle \lambda_{i_{0}}, \lambda_{i_{1}}, \ldots, \lambda_{i_{m-1}}, \lambda'_{\text{top(uclock)}} = \text{link} \rangle & \text{if } \text{top(uclock)} \neq i_{m-1}
\end{cases}
\]
Once again the valid suffix is gone. In other words the mergers which have occurred in the failed branches and in which this particular node has participated are undone. More precise the node's contribution to those mergers is undone (it takes two to make a merger).

6.6 Summary

This chapter describes the unification process in detail. The four major components of the unification process are discussed — current grammar branch selection, current constituent selection, constituent–branch unification and handling of undoing upon backtracking.

The branch–selection algorithm follows a simple depth–first, left to right traversal of the grammar AND–OR tree (cf. Sec.6.2).

The particular heuristics for the definition of the order of the constituent tree induced by the cset structure of the unified FD is described in Sec.6.3. This heuristics has proved empirically to improve the execution speed on the SURGE benchmark without affecting the correctness of the results.

The major design decision has been to optimize the undoing operation to make backtracking as efficient as possible. This has led to an implementation of modifications based on a clock mechanism for the FD graph, which is described in Sec.6.5. Merging of nodes follows the “passive merger” method of linking equivalent nodes using clock–labelled links into composite nodes.

The evaluation of this design decision in terms of performance is complex and it is the main topic of the next chapter.
Chapter 7

Evaluation

7.1 Overview

The evaluation process can be divided in two parts:

1. Comparing CFUF vs. LispFUF. The existing Lisp implementation of FUF (LispFUF) has served as the benchmark reference for the new implementation. The grammar used is the SURGE English Grammar written by Elhadad [4], [5] along with 421 example inputs provided with the FUF/SURGE public distribution. The SURGE grammar has about 5600 branches and 1600 alts (i.e., choice points).

Most inputs unified with SURGE produce the same result both in LispFUF and CFUF, but some still differ. There are two reasons for the differences — the different order of constituent unification and bugs in LispFUF and/or CFUF. While the latter happens unintentionally, the former is the consequence of my observations. It appears that the default order used in LispFUF gives worse results than the order I am employing (see section 6.3). In particular it fixes the 'its' bug — a number of examples in LispFUF produce *its* where *his, her, their* ... is expected. Also the default order results in more failures — for the set I have used — 22 failures out of 421 inputs vs. 19 out of 421\(^1\).

Section 7.2 brings the results of comparing CFUF vs. LispFUF.

\(^1\)A failure in the default order does not necessarily mean that the same input will fail in my order.
CHAPTER 7. EVALUATION

2. Profiling various CFUF implementation aspects. Here I am trying to justify some key CFUF implementation decisions, like choosing passive merger vs. active merger (see 3.3.1), etc... The same set of SURGE inputs as in section 7.2 is used. Again, only successful unifications counted (the total of 402). The profiling is turned on during unification only, i.e., only then do I collect the statistics (because unification takes the lion share of overall run time).

See section 7.3 for the discussion.

7.2 CFUF vs LispFUF

Definition 24 Unification time.
Let \( fd \) and \( gr \) be an FD and a grammar respectively, then \( utime_L(fd, gr) \) \( (utime_C(fd, gr)) \) denotes the time it takes to unify the input \( fd \) with the grammar \( gr \) in LispFUF (CFUF).

Definition 25 Time ratio.
Let \( fd \) and \( gr \) be an FD and a grammar respectively, then \( ratio(fd, gr) \) is the fraction \( \frac{utime_L(fd, gr)}{utime_C(fd, gr)} \).

I will use the term time ratio when I am not interested in actual numbers, but rather in general properties of \( ratio(fd, gr) \), when \( gr \) is fixed (SURGE for instance) and \( fd \) iterates over a large set of “real life” inputs.

I have used two lisp implementations to run LispFUF — GCL (GNU Common Lisp) Version(2.2.2) and Allegro CL 4.3. CFUF was compiled with egcs–2.90.29 980515 (egcs–1.0.3 release).

Hypothesis 1 Time ratio tends to grow with the growth of the number of backtracking points.

In other words, I wish to show that the more backtracking occurs the more efficient CFUF gets vs LispFUF.

To verify the hypothesis I ran both CFUF and LispFUF on SURGE with 421 input fds. Every fd was unified with SURGE 10 times and the minimum of the 10 tries was taken. Also to minimize the affect of Garbage Collector (GC) in LispFUF, it was run explicitly before every call to unification (in CFUF the GC of scm is never invoked during unification).
Figure 7.1 contains the graphs of time ratio (Y-axis) as function of the number of backtracking points (X-axis). The top graph corresponds to ACL total, the middle one — to GCL total and the bottom — to ACL non-gc times.

Since ACL reports both pure algorithmic time (non-gc) and the total time I have included both of them. Notice the huge impact of the GC for inputs requiring over 400 backtracking points in ACL despite the abundance of memory — the machine on which the test was performed had 64M of which at the time of the test 55M were available.

Table 7.1 lists the greatest times (in seconds) of successful unifications sorted by CFUF times. Once again the LispFUF times are presented by three groups — ACL non-gc, ACL total and GCL total times.

**Hypothesis 2** Time ratio tends to grow with the growth of the input fd depth.

By depth I mean the depth of the graph $\tilde{G}_{fd}$ used to represent the fd.
<table>
<thead>
<tr>
<th>input</th>
<th>number of backtracking points</th>
<th>LispFUF ACL non-gc</th>
<th>ACL total</th>
<th>GCL total</th>
<th>CFUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>t423</td>
<td>813</td>
<td>4.87</td>
<td>57.51</td>
<td>12.78</td>
<td>0.38</td>
</tr>
<tr>
<td>c40</td>
<td>788</td>
<td>2.28</td>
<td>38.55</td>
<td>4.65</td>
<td>0.36</td>
</tr>
<tr>
<td>c26</td>
<td>899</td>
<td>1.66</td>
<td>25.11</td>
<td>4.33</td>
<td>0.34</td>
</tr>
<tr>
<td>t420</td>
<td>745</td>
<td>4.47</td>
<td>43.91</td>
<td>11.74</td>
<td>0.33</td>
</tr>
<tr>
<td>c14</td>
<td>833</td>
<td>1.24</td>
<td>24.09</td>
<td>2.77</td>
<td>0.32</td>
</tr>
<tr>
<td>c27</td>
<td>751</td>
<td>1.14</td>
<td>10.15</td>
<td>2.58</td>
<td>0.30</td>
</tr>
<tr>
<td>c30</td>
<td>696</td>
<td>1.88</td>
<td>22.00</td>
<td>4.77</td>
<td>0.29</td>
</tr>
<tr>
<td>c16</td>
<td>684</td>
<td>2.43</td>
<td>30.93</td>
<td>6.40</td>
<td>0.28</td>
</tr>
<tr>
<td>c6</td>
<td>676</td>
<td>1.07</td>
<td>18.93</td>
<td>2.46</td>
<td>0.27</td>
</tr>
<tr>
<td>c24</td>
<td>741</td>
<td>1.33</td>
<td>8.56</td>
<td>3.27</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Table 7.1: CFUF largest times on the input set (on Linux i586)

To verify the hypothesis the following technique is used — a simple sentence, like “John loves Mary” is wrapped in “I think that” several times to yield the sequence of sentences with the depths of the respective fds growing in direct proportion to the number of “I think that”:

0 nesting: John loves Mary.

1 nesting: I think that John loves Mary.

2 nesting: I think that I think that John loves Mary.

n nesting: \[ I \text{think that} \ldots I \text{think that} \text{John loves Mary.} \]

The respective fds are then unified with SURGE in CFUF and LispFUF (the technique remains unchanged — each fd is unified 10 times, GC is called prior every call to unification).

As expected, the deeper the nesting the more backtracking points it takes to get the result. In fact each “I think that” adds 110 more backtracking points.

Figure 7.2 contains the graphs of time ratio as function of the number of backtracking points for ACL total, GCL total and ACL non-gc times.
Figure 7.2: Nesting: LispFUF vs. CFUF (on Linux i586) X-axis is the number of backtracking points, Y-axis is speedup factor. Benchmark is on the set of nested clauses “I think that ...”
Table 7.2: Nesting times (on Linux i586)

Table 7.2 contains the times for the nesting test sorted by CFUF times. A test $d_i$ correspond to the nesting $i^2$.

Note, that $u_{\text{time}}(d_i, \textit{SU RGE})$ grows more or less linearly with $i$, meaning that linear increase in depth is likely to produce linear increase in time in CFUF.

To get as objective picture as possible I have performed all the above tests on sparc Ultra-4 machine running Solaris OS. CFUF has been compiled with the same compiler as on Linux, LispFUF has been run on ACL 4.3.1, no GCL this time. Figures 7.3 and 7.4 show the graphs for hypotheses 1 and 2 respectively and Tables 7.3 and 7.4 show the times.

### 7.3 CFUF Profiling

The following implementation components will be evaluated in the following sections:

- Composite nodes (passive vs active merger).
- Valid suffix of type/link sequence.
- Composite node size/cancelled nodes percentage.

---

$^2$ACL on Linux was dropping dead with “Segmentation fault” if I dared to go beyond nesting 6.
Figure 7.3: LispFUF vs CFUF (on Solaris sparc Ultra-4) X-axis is the number of backtracking points; Y-axis is the speedup

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of Backtracking Points</th>
<th>LispFUF (ACL) non-gc</th>
<th>LispFUF (ACL) total</th>
<th>CFUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1423</td>
<td>813</td>
<td>1.95</td>
<td>15.45</td>
<td>0.21</td>
</tr>
<tr>
<td>c40</td>
<td>788</td>
<td>0.91</td>
<td>10.39</td>
<td>0.20</td>
</tr>
<tr>
<td>c26</td>
<td>899</td>
<td>0.67</td>
<td>10.81</td>
<td>0.19</td>
</tr>
<tr>
<td>1420</td>
<td>745</td>
<td>1.82</td>
<td>12.51</td>
<td>0.18</td>
</tr>
<tr>
<td>c14</td>
<td>833</td>
<td>0.52</td>
<td>7.99</td>
<td>0.17</td>
</tr>
<tr>
<td>c30</td>
<td>696</td>
<td>0.77</td>
<td>9.42</td>
<td>0.16</td>
</tr>
<tr>
<td>c27</td>
<td>751</td>
<td>0.45</td>
<td>4.34</td>
<td>0.16</td>
</tr>
<tr>
<td>c24</td>
<td>741</td>
<td>0.52</td>
<td>4.47</td>
<td>0.15</td>
</tr>
<tr>
<td>c22</td>
<td>671</td>
<td>0.49</td>
<td>4.26</td>
<td>0.15</td>
</tr>
<tr>
<td>c16</td>
<td>684</td>
<td>0.98</td>
<td>10.95</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 7.3: CFUF greatest times (on Solaris sparc Ultra-4)
CHAPTER 7. EVALUATION

Figure 7.4: Nesting: LispFUF vs. CFUF (on Solaris sparc Ultra-4) X-axis is the number of backtracking points; Y-axis is speedup.

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of Backtracking Points</th>
<th>LispFUF (ACL)</th>
<th>CFUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_{10}$</td>
<td>1250</td>
<td>2.98</td>
<td>27.63</td>
</tr>
<tr>
<td>$d_9$</td>
<td>1140</td>
<td>1.78</td>
<td>24.35</td>
</tr>
<tr>
<td>$d_8$</td>
<td>1030</td>
<td>1.49</td>
<td>21.36</td>
</tr>
<tr>
<td>$d_7$</td>
<td>920</td>
<td>1.21</td>
<td>19.56</td>
</tr>
<tr>
<td>$d_6$</td>
<td>810</td>
<td>0.98</td>
<td>13.38</td>
</tr>
<tr>
<td>$d_5$</td>
<td>700</td>
<td>0.76</td>
<td>10.88</td>
</tr>
<tr>
<td>$d_4$</td>
<td>590</td>
<td>0.59</td>
<td>8.55</td>
</tr>
<tr>
<td>$d_3$</td>
<td>480</td>
<td>0.41</td>
<td>3.83</td>
</tr>
<tr>
<td>$d_2$</td>
<td>370</td>
<td>0.28</td>
<td>1.65</td>
</tr>
<tr>
<td>$d_1$</td>
<td>260</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>$d_0$</td>
<td>150</td>
<td>0.06</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Table 7.4: Nesting times (on Solaris sparc Ultra-4)
Ordinarily, a component will be characterized by some random variable $RV : \{i\}_{i=0}^\infty \rightarrow [0..1]$. For example, let $N \in V_{\sim_G}$, then the random variable $RV_{\text{suffix}}$ of the valid suffix of the node type sequence is its length — $|\text{suffix}(\text{typeseq}(N))|$, in other words $RV_{\text{suffix}}(i)$ is the probability that when a node type is accessed the length of the valid suffix is $i$.

The random variables are approximated for every tested input by calculating the frequencies of the appropriate events.

From now on, the term **random variable** will refer to the random variable of some implementation component.

**Definition 26 Random variable upper bound.**

For every random variable $RV$, there exists such an integer $\max(RV) \geq 0$, called the **upper bound** of the random variable $RV$, such that $\forall i > \max(RV) \ (RV(i) = 0)$.

The smaller the upper bound of a random variable the more efficient the respective component turns out to be. For example, let $RV$ be the random variable of valid suffix (type sequence or link sequence, whatever). It is clear, that the less the length of valid suffix the faster the access to the node type or link. Thus if during the tests I discover that $\max(RV) = 15$, this will mean that the implementation of backtracking is inefficient, because it causes some nodes to have too long valid suffixes, which in turn slows down type/link access.

To know the upper bound is not enough to decide whether the implementation is efficient or not. The probability of the upper bound and the values close to it is important as well.

The general format of each subsequent section is:

**RV:** Defines the random variable $RV$ for the component evaluated.

**Worst case:** Lists the 10 worst cases in a table. Usually there are two tables. One is sorted in decreasing order by the probabilities of $\max(RV)$ or a value close to it, another one — in increasing order by the probabilities of $RV = 0$ or $RV = 1$. The former is supposed to demonstrate that “bad” instances are rare and the latter — that “good” instances occur very often.

**Graphs:** The graphs of $RV(i)$ as a function of the number of visited branches are given for a couple of values of $i$. If there is a dependency between the two, it should be revealed by the corresponding graph.
Conclusion: A conclusion as to efficiency (or inefficiency) of the evaluated component is made, based on the results of the observations.

7.3.1 Type Sequence

The type sequence of a node in the internal graph is the stack of type tags that are attached to the node as unification proceeds. The clock mechanism defined in Sect.6.5 basically attaches to each node a stack of types. The basic query on a node is “what is the type of this node?” The time to execute this query depends on the length of the type stack. We evaluate the overhead induced by the use of this mechanism in this section.

Recall from section 6.5 that the type of an individual node \( n \in V \) is the last value of \( vprefix(typeseq(n)) \). This means, that the valid suffix of \( typeseq(n) \) has to be discarded, in other words time to execute \( type(n) \) is directly proportional to the length of \( vsuffix(typeseq(n)) \). Naturally it is important to explore the behavior of \( vsuffix(typeseq(n)) \), because when looking for the type of a node, the interpreter starts from the end of the stack and progresses backwards until the valid suffix has been traversed.

**Definition 27** Random variable \( RV_{tsuf} \) denotes the length of \( vsuffix(typeseq(n)) \) for an individual node \( n \in V \).

Random variable \( RV_{tseq} \) denotes the length of \( typeseq(n) \) for an individual node \( n \in V \).

**Worst Case** \( RV_{tsuf} \) It appears that \( \max(RV_{tsuf}) = 2 \), in other words for every visited node no valid suffix ever contains more than two elements, see Table 7.5.

In most of the tests less than 0.1% of the calls to \( type(n) \) (for any node \( n \)) encounter the valid suffix of two elements. Also, in more than 98% of the calls in almost all the tests there is no valid suffix at all!

**Graphs** \( RV_{tsuf} \) Figures 7.5, 7.6 and 7.7 show the graphs of \( RV_{tsuf}(i) \) as functions of the number of visited branches respectively. According to the graphs, \( RV_{tsuf}(0) \) tends to stay in the vicinity of 0.99, while \( RV_{tsuf}(1) \) stays below 0.01. Further experiments, however, indicate that \( RV_{tsuf}(0) \) tends to grow with the growth of the number of visited branches, which is good. The growth, though, slows down constantly.
<table>
<thead>
<tr>
<th>test</th>
<th>number of branches visited</th>
<th>number of typeseq accesses</th>
<th>(RV_{tsuf}(0))</th>
<th>(RV_{tsuf}(1))</th>
<th>(RV_{tsuf}(2))</th>
<th>(RV_{tsuf}(&gt; 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1192</td>
<td>480</td>
<td>8904</td>
<td>0.956649</td>
<td>0.041105</td>
<td>0.0032246</td>
<td>0.000000</td>
</tr>
<tr>
<td>1176</td>
<td>317</td>
<td>6477</td>
<td>0.947255</td>
<td>0.052427</td>
<td>0.003161</td>
<td>0.000000</td>
</tr>
<tr>
<td>1179</td>
<td>317</td>
<td>6477</td>
<td>0.956649</td>
<td>0.041105</td>
<td>0.0032246</td>
<td>0.000000</td>
</tr>
<tr>
<td>1178</td>
<td>183</td>
<td>3979</td>
<td>0.957778</td>
<td>0.041211</td>
<td>0.003011</td>
<td>0.000000</td>
</tr>
<tr>
<td>1181</td>
<td>293</td>
<td>6045</td>
<td>0.947229</td>
<td>0.050786</td>
<td>0.001985</td>
<td>0.000000</td>
</tr>
<tr>
<td>1176</td>
<td>195</td>
<td>4125</td>
<td>0.956649</td>
<td>0.041105</td>
<td>0.0032246</td>
<td>0.000000</td>
</tr>
<tr>
<td>1180</td>
<td>179</td>
<td>3869</td>
<td>0.958646</td>
<td>0.039804</td>
<td>0.001551</td>
<td>0.000000</td>
</tr>
<tr>
<td>1176</td>
<td>179</td>
<td>3869</td>
<td>0.958646</td>
<td>0.039804</td>
<td>0.001551</td>
<td>0.000000</td>
</tr>
<tr>
<td>1205</td>
<td>186</td>
<td>4622</td>
<td>0.973172</td>
<td>0.025530</td>
<td>0.001298</td>
<td>0.000000</td>
</tr>
<tr>
<td>12bis</td>
<td>28</td>
<td>1245</td>
<td>0.989906</td>
<td>0.009960</td>
<td>0.000534</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

Table 7.5: The largest values of \(RV_{tsuf}(2)\). Indicates the probability that 0, 1 or 2 type tags must be checked until the valid type of a node is found.

<table>
<thead>
<tr>
<th>test</th>
<th>number of branches visited</th>
<th>number of typeseq accesses</th>
<th>(RV_{tsuf}(0))</th>
<th>(RV_{tsuf}(1))</th>
<th>(RV_{tsuf}(2))</th>
<th>(RV_{tsuf}(&gt; 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1175</td>
<td>317</td>
<td>6477</td>
<td>0.942112</td>
<td>0.055427</td>
<td>0.002161</td>
<td>0.000000</td>
</tr>
<tr>
<td>1181</td>
<td>293</td>
<td>6045</td>
<td>0.947229</td>
<td>0.050786</td>
<td>0.001985</td>
<td>0.000000</td>
</tr>
<tr>
<td>1176</td>
<td>195</td>
<td>4125</td>
<td>0.956649</td>
<td>0.041105</td>
<td>0.0032246</td>
<td>0.000000</td>
</tr>
<tr>
<td>1192</td>
<td>480</td>
<td>8904</td>
<td>0.956649</td>
<td>0.041105</td>
<td>0.0032246</td>
<td>0.000000</td>
</tr>
<tr>
<td>1178</td>
<td>183</td>
<td>3979</td>
<td>0.957778</td>
<td>0.040211</td>
<td>0.002011</td>
<td>0.000000</td>
</tr>
<tr>
<td>1174</td>
<td>183</td>
<td>3974</td>
<td>0.957778</td>
<td>0.040010</td>
<td>0.002013</td>
<td>0.000000</td>
</tr>
<tr>
<td>1180</td>
<td>179</td>
<td>3869</td>
<td>0.958646</td>
<td>0.039804</td>
<td>0.001551</td>
<td>0.000000</td>
</tr>
<tr>
<td>1179</td>
<td>179</td>
<td>3869</td>
<td>0.958646</td>
<td>0.039804</td>
<td>0.001551</td>
<td>0.000000</td>
</tr>
<tr>
<td>1205</td>
<td>186</td>
<td>4622</td>
<td>0.973172</td>
<td>0.025530</td>
<td>0.001298</td>
<td>0.000000</td>
</tr>
<tr>
<td>e20</td>
<td>411</td>
<td>13206</td>
<td>0.988263</td>
<td>0.011283</td>
<td>0.000454</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

Table 7.6: The smallest values of \(RV_{tsuf}(0)\)
Figure 7.5: $RV_{tsuf}(0)$: Probability that the first type checked is the valid type

**Worst Case $RV_{seq}$** What about the total length of $typeseq(n)$? Let us ignore the special types, then $typeseq(n)$ is bound by a constant determined by the type hierarchy used during the unification. Indeed, if $typeseq(n) = \langle \tau_i_0, \tau_i_1, \ldots, \tau_i_{k-1} \rangle$, then in CFUF $\tau_i_0 \succ \tau_i_1 \succ \cdots \succ \tau_i_{k-3}$ (the optimized version). Even if $\tau_i_0 = nil$ and $\tau_i_1 = any$ it still leaves us with $k - 3$ different types forming a linear order or a **type chain** with each type being more specialized than the previous one. But that number is bound by the length of the longest type chain in the type hierarchy.

The type hierarchy used with the SURGE grammar has no chains longer than 9 types. That gives us the upper bound for $typeseq(n)$.

My findings show, that $\max(RV_{seq}) = 6$, which is pretty close to the upper bound. Also, in most of the tests, at least 80% of the type lookups encounter $|typeseq(n)| = 1$ and 20% — $|typeseq(n)| = 2$. 
7.3.2 Link Sequence

In a manner similar to the way the type is encoded in nodes, links in the FD graph are annotated with clock information. This information takes the form of a stack of clock ticks that indicate for which clock values the link is valid (cf. Sect.6.5). The basic query on a node we address here is “what is the child of this node under feature f?” The time to execute this query depends on the number of nodes aggregated into a composite node in the quotient graph and on the number of clock ticks attached to each outgoing link from each individual node in the composite node. We evaluate the overhead induced by the use of this mechanism for a single link in this section (either between an individual node and its child or between an individual node and another equivalent individual node within a composite node). In the next section, we evaluate the number of individual nodes in a composite node.

Recall from section 6.5 that the link of an individual node \( n \in V \) is the last value of \( vprefix(linkseq(n)) \). This means, that the valid suffix of \( linkseq(n) \) has to be discarded, in other words time to retrieve \( link(n) \) (which
is a pair \( (\text{cnxt}(n), \text{ufnext}(n), ) \) is directly proportional to the length of \( \text{vsuf fix}(\text{linkseq}(n)) \). Naturally it is important to explore the behavior of \( \text{vsuf fix}(\text{linkseq}(n)) \).

**Definition 28** Random variable \( \text{RV}_{\text{vsuf}} \) denotes the length of \( \text{vsuf fix}(\text{linkseq}(n)) \) for an individual node \( n \in V \).

Random variable \( \text{RV}_{\text{seq}} \) denotes the length of \( \text{linkseq}(n) \) for an individual node \( n \in V \).

**Worst Case** \( \text{RV}_{\text{vsuf}} \) It appears that \( \max(\text{RV}_{\text{vsuf}}) = 3 \). However, there is only one test for which \( \text{RV}_{\text{vsuf}}(3) \neq 0 \), namely “t420” with \( \text{RV}_{\text{vsuf}}(3) = 0.000013 \) (a single occasion out of 77208) and there are only 20 out of 402 tests with \( \text{RV}_{\text{vsuf}}(2) \neq 0 \), where \( \text{RV}_{\text{vsuf}}(2) \) ranges from 0.000826 to 0.000016. For the rest of the tests \( \max(\text{RV}_{\text{vsuf}}) = 1 \), in other words for every visited node no valid suffix every contains more than one element, see Tables 7.7 and 7.8. It follows, that less than 2% of the \( \text{linkseq} \) accesses hit a valid suffix of the unit length, the figure is even less for most of the tests — about
Table 7.7: The largest values of $RV_{\text{isuf}}$: Probability that a valid link is found in 1, 2, 3 or more clock tag checks

<table>
<thead>
<tr>
<th>test</th>
<th>number of branches visited</th>
<th>number of linkseq accesses</th>
<th>$RV_{\text{isuf}}(0)$</th>
<th>$RV_{\text{isuf}}(1)$</th>
<th>$RV_{\text{isuf}}(2)$</th>
<th>$RV_{\text{isuf}}(&gt;2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1176</td>
<td>195</td>
<td>7773</td>
<td>0.979416</td>
<td>0.019941</td>
<td>0.000643</td>
<td>0.000000</td>
</tr>
<tr>
<td>1180</td>
<td>179</td>
<td>7265</td>
<td>0.980729</td>
<td>0.018905</td>
<td>0.000825</td>
<td>0.000000</td>
</tr>
<tr>
<td>1179</td>
<td>179</td>
<td>7269</td>
<td>0.980241</td>
<td>0.018933</td>
<td>0.000823</td>
<td>0.000000</td>
</tr>
<tr>
<td>1178</td>
<td>183</td>
<td>7461</td>
<td>0.980566</td>
<td>0.018764</td>
<td>0.000870</td>
<td>0.000000</td>
</tr>
<tr>
<td>1174</td>
<td>183</td>
<td>7477</td>
<td>0.980741</td>
<td>0.018590</td>
<td>0.000669</td>
<td>0.000000</td>
</tr>
<tr>
<td>1175</td>
<td>317</td>
<td>11976</td>
<td>0.983049</td>
<td>0.016333</td>
<td>0.000418</td>
<td>0.000000</td>
</tr>
<tr>
<td>1181</td>
<td>293</td>
<td>11231</td>
<td>0.983083</td>
<td>0.016383</td>
<td>0.000534</td>
<td>0.000000</td>
</tr>
<tr>
<td>1192</td>
<td>480</td>
<td>15972</td>
<td>0.984786</td>
<td>0.015214</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1205</td>
<td>186</td>
<td>8750</td>
<td>0.987771</td>
<td>0.012229</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1391</td>
<td>40</td>
<td>1160</td>
<td>0.993966</td>
<td>0.006034</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

Table 7.8: The smallest values of $RV_{\text{isuf}}(0)$

<table>
<thead>
<tr>
<th>test</th>
<th>number of branches visited</th>
<th>number of linkseq accesses</th>
<th>$RV_{\text{isuf}}(0)$</th>
<th>$RV_{\text{isuf}}(1)$</th>
<th>$RV_{\text{isuf}}(2)$</th>
<th>$RV_{\text{isuf}}(&gt;2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1176</td>
<td>195</td>
<td>7773</td>
<td>0.979416</td>
<td>0.019941</td>
<td>0.000643</td>
<td>0.000000</td>
</tr>
<tr>
<td>1180</td>
<td>179</td>
<td>7265</td>
<td>0.980729</td>
<td>0.018905</td>
<td>0.000825</td>
<td>0.000000</td>
</tr>
<tr>
<td>1179</td>
<td>179</td>
<td>7269</td>
<td>0.980241</td>
<td>0.018933</td>
<td>0.000823</td>
<td>0.000000</td>
</tr>
<tr>
<td>1178</td>
<td>183</td>
<td>7461</td>
<td>0.980566</td>
<td>0.018764</td>
<td>0.000870</td>
<td>0.000000</td>
</tr>
<tr>
<td>1174</td>
<td>183</td>
<td>7477</td>
<td>0.980741</td>
<td>0.018590</td>
<td>0.000669</td>
<td>0.000000</td>
</tr>
<tr>
<td>1175</td>
<td>317</td>
<td>11976</td>
<td>0.983049</td>
<td>0.016333</td>
<td>0.000418</td>
<td>0.000000</td>
</tr>
<tr>
<td>1181</td>
<td>293</td>
<td>11231</td>
<td>0.983083</td>
<td>0.016383</td>
<td>0.000534</td>
<td>0.000000</td>
</tr>
<tr>
<td>1192</td>
<td>480</td>
<td>15972</td>
<td>0.984786</td>
<td>0.015214</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1205</td>
<td>186</td>
<td>8750</td>
<td>0.987771</td>
<td>0.012229</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1391</td>
<td>40</td>
<td>1160</td>
<td>0.993966</td>
<td>0.006034</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
0.6%. At least 97.9% (generally 99.3%) of the accesses do not encounter the valid suffix at all (in other words $\text{linkseq}(n) = \text{prefix}(\text{linkseq}(n))$).

**Graphs** $RV_{i, suf}$  Figures 7.8, 7.9 and 7.10 show the graphs of $RV_{i, suf}(i)$ as functions of the number of visited branches respectively. Further experiments indicate that $RV_{i, suf}(0)$ tends to grow with the growth of the number of visited branches, which is good. The growth, though, slows down constantly.

**Worst Case** $RV_{i, seq}$  For the inputs I have tested, $\max(RV_{i, seq}) = 17$, however, such a long link sequence is a rarity (see Table 7.9). At least 70% of the link lookups hit a link sequence of one element and less than 22% — of two elements.

**Graphs** $RV_{i, seq}$  Figures 7.11, 7.12 and 7.13 show the graphs of $RV_{i, seq}(i)$ as functions of the number of visited branches respectively.
Figure 7.9: $RV_{isuf}(1)$: Probability that a valid link is found in 2 clock tag checks

<table>
<thead>
<tr>
<th>$i$</th>
<th>Max $RV_{isuf}(t)$</th>
<th>Average $RV_{isuf}(t)$</th>
<th>the number of tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0.000211</td>
<td>0.000001</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0.000289</td>
<td>0.000001</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>0.000555</td>
<td>0.000002</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0.000555</td>
<td>0.000003</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>0.000587</td>
<td>0.000008</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>0.000691</td>
<td>0.000010</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>0.000658</td>
<td>0.000013</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>0.001616</td>
<td>0.000026</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>0.001887</td>
<td>0.000045</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>0.001954</td>
<td>0.000102</td>
<td>70</td>
</tr>
<tr>
<td>7</td>
<td>0.001887</td>
<td>0.000135</td>
<td>91</td>
</tr>
<tr>
<td>6</td>
<td>0.002909</td>
<td>0.000284</td>
<td>123</td>
</tr>
<tr>
<td>5</td>
<td>0.007294</td>
<td>0.001244</td>
<td>202</td>
</tr>
<tr>
<td>4</td>
<td>0.089111</td>
<td>0.009484</td>
<td>316</td>
</tr>
</tbody>
</table>

Table 7.9: The largest values of $RV_{isuf}(i)$
Figure 7.10: $RV_{lsuf}(2)$: Probability that a valid link is found in 3 clock tag checks

Figure 7.11: $RV_{lseq}(1)$
Figure 7.12: $RV_{lseq}(2)$

Figure 7.13: $RV_{lseq}(3)$
Passive Merger vs Active Merger

In the following discussion I ignore the overhead of backtracking when traversing links in the internal FD structure.

The passive and active mergers are described in 3.3.1. The goal is to compare these merger techniques with respect to the time it takes to perform the six major operations concerning nodes, see Table 3.2 in section 3.3.1. Since I have not actually implemented the active merger, I do not present any experimental observations of it, however, I can imagine the properties of the ideal implementation. My assumptions for such an implementation are:

1. Active node merger, as well as its undo, involves physical movement of node children and update of references.

2. Accessing node type is immediate.

3. Child node lookup, both successful and not, takes one hash table access.

As can be seen, merging two composite nodes and undoing their merger is clearly faster in the passive merger than in the active merger, since the former requires changing four pointers only (update the c–neighbors and uf–neighbours of the two representatives).

The more common operations, like get/set node type and find/add child node, however, are slower in the passive merger, since a composite node can be actually a group of individual nodes. This means:

1. The group representative must first be located in order to get the node type. This involves dereferencing the ufnext pointers of the individual nodes.

2. A child node lookup involves iteration on the individual nodes comprising the composite one. Moreover, all the individual nodes must be visited in the event of unsuccessful lookup.

**Definition 29** Random variable RV\(_{uf}\) denotes the number of ufnext pointers, which must be dereferenced to access a node type. Random variable RV\(_{i}\) denote the number of individual nodes which must be visited to lookup a node child or, which is the same, the number of hash table accesses to lookup the child.
### Table 7.10: The greatest values of $RV_c(6)$

<table>
<thead>
<tr>
<th>test</th>
<th>number of branches visited</th>
<th>number of child lookups</th>
<th>$RV_c(1)$</th>
<th>$RV_c(2)$</th>
<th>$RV_c(3)$</th>
<th>$RV_c(4)$</th>
<th>$RV_c(5)$</th>
<th>$RV_c(6)$</th>
<th>$RV_c(&gt; 6)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>t129</td>
<td>148</td>
<td>1290</td>
<td>0.710600</td>
<td>0.021667</td>
<td>0.006667</td>
<td>0.003250</td>
<td>0.003333</td>
<td>0.133333</td>
<td>0.000000</td>
</tr>
<tr>
<td>t132</td>
<td>150</td>
<td>1215</td>
<td>0.702463</td>
<td>0.026373</td>
<td>0.005473</td>
<td>0.002930</td>
<td>0.001568</td>
<td>0.131987</td>
<td>0.000000</td>
</tr>
<tr>
<td>t196is</td>
<td>225</td>
<td>2066</td>
<td>0.673666</td>
<td>0.032433</td>
<td>0.158761</td>
<td>0.010165</td>
<td>0.008712</td>
<td>0.003533</td>
<td>0.000000</td>
</tr>
<tr>
<td>t131</td>
<td>201</td>
<td>1749</td>
<td>0.718835</td>
<td>0.016099</td>
<td>0.061750</td>
<td>0.108626</td>
<td>0.010863</td>
<td>0.001481</td>
<td>0.000000</td>
</tr>
<tr>
<td>9</td>
<td>110</td>
<td>9799</td>
<td>0.671707</td>
<td>0.122092</td>
<td>0.072200</td>
<td>0.051091</td>
<td>0.043104</td>
<td>0.001393</td>
<td>0.000000</td>
</tr>
<tr>
<td>t128</td>
<td>206</td>
<td>1758</td>
<td>0.716904</td>
<td>0.015586</td>
<td>0.066091</td>
<td>0.113199</td>
<td>0.001183</td>
<td>0.001013</td>
<td>0.000000</td>
</tr>
<tr>
<td>t134</td>
<td>206</td>
<td>1808</td>
<td>0.790460</td>
<td>0.017146</td>
<td>0.063053</td>
<td>0.023783</td>
<td>0.010000</td>
<td>0.089409</td>
<td>0.000000</td>
</tr>
<tr>
<td>111</td>
<td>207</td>
<td>1812</td>
<td>0.794702</td>
<td>0.017660</td>
<td>0.064770</td>
<td>0.023731</td>
<td>0.010486</td>
<td>0.088852</td>
<td>0.000000</td>
</tr>
<tr>
<td>110</td>
<td>207</td>
<td>1812</td>
<td>0.794702</td>
<td>0.017660</td>
<td>0.064770</td>
<td>0.023731</td>
<td>0.010486</td>
<td>0.088852</td>
<td>0.000000</td>
</tr>
<tr>
<td>t30</td>
<td>210</td>
<td>1829</td>
<td>0.797157</td>
<td>0.015300</td>
<td>0.066010</td>
<td>0.023510</td>
<td>0.010388</td>
<td>0.088026</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

### Table 7.11: The smallest values of $RV_c(1)$

Note that $RV_{a,f}(0) = 0$, because at least one $ufnext$ pointer is always dereferenced (to make sure that the individual node is the representative).

To argue the superiority (or inferiority) of the passive merger over the active, one has to explore the properties of $RV_{a,f}$ and $RV_c$.

**Worst Case $RV_c$** There is only one test (“t364”) with $max(RV_c) = 8$, where $RV_c(8) = 0.023259$ and 16 with $max(RV_c) = 7$, where $RV_c(7)$ ranges from 0.067539 to 0.001401. In other words, no more than 68 out of 1000 child lookups will take 7 hash table accesses for the worst of those tests. The rest of the tests have $max(RV_c) = 6$, see Table 7.10.

As you can see, ordinarily $RV_c(6) < 0.1$, but still there are about 10% of child lookups which end up with 6 hash table accesses, as opposed to one for ideal active merger.
Table 7.11 lists the smallest values of $RV_c(1)$. The good news, is that almost always more than 60% of child lookups will be as efficient as in ideal active merger.

**Graphs RV\(_c\)** Figures 7.14, 7.15, 7.16 and 7.17 show the graphs of $RV_c(i)$ as functions of the number of visited branches respectively. According to the graphs, $RV_c(1)$ has a tendency to decrease with the growth of the number of visited branches, while $RV_c(2)$ — to increase. This result is quite logical, as usually the more branches are visited the more complex input is and as such, it is likely to have more large composite nodes. Further experiments, however, show that $RV_c(1)$ starts to increase back after about 1100 visited branches. I find it difficult to explain at the moment.

**Worst Case RV\(_{uf}\)** There are 5 tests ("t405", "t27bis", "t26", "t213" and "mine3") with $\max(RV_{uf}) = 4$. In each of those tests there is a single occasion of a type lookup requiring participation of 4 $ufnext$ pointers to complete. The rest of the tests have $\max(RV_{uf}) = 3$, see Table 7.12 and 7.13.
Figure 7.15: $RV_c(2)$

Figure 7.16: $RV_c(3)$
Figure 7.17: $RV_c(4)$, $RV_c(5)$ and $RV_c(6)$

<table>
<thead>
<tr>
<th>test</th>
<th>number of branches visited</th>
<th>number of type lookups</th>
<th>$RV_{uf}(1)$</th>
<th>$RV_{uf}(2)$</th>
<th>$RV_{uf}(3)$</th>
<th>$RV_{uf}(4)$</th>
<th>$RV_{uf}(&gt;4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>t402</td>
<td>214</td>
<td>5417</td>
<td>0.593317</td>
<td>0.342071</td>
<td>0.004611</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t120</td>
<td>88</td>
<td>2058</td>
<td>0.750438</td>
<td>0.203527</td>
<td>0.046255</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t403</td>
<td>356</td>
<td>10002</td>
<td>0.623275</td>
<td>0.348630</td>
<td>0.028094</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t388s</td>
<td>199</td>
<td>6200</td>
<td>0.703333</td>
<td>0.271774</td>
<td>0.027003</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t113</td>
<td>147</td>
<td>3632</td>
<td>0.698948</td>
<td>0.278666</td>
<td>0.029866</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t3iter</td>
<td>203</td>
<td>6289</td>
<td>0.707108</td>
<td>0.266338</td>
<td>0.026554</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t26</td>
<td>584</td>
<td>17002</td>
<td>0.663675</td>
<td>0.310590</td>
<td>0.025679</td>
<td>0.000057</td>
<td>0.00000</td>
</tr>
<tr>
<td>t27</td>
<td>639</td>
<td>39203</td>
<td>0.661615</td>
<td>0.313597</td>
<td>0.024178</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t52</td>
<td>154</td>
<td>3876</td>
<td>0.691796</td>
<td>0.283634</td>
<td>0.024161</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>t115</td>
<td>147</td>
<td>3628</td>
<td>0.697494</td>
<td>0.278090</td>
<td>0.024117</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

Table 7.12: The greatest values of $RV_{uf}(3)$
It follows from the observations, that usually less than 3% of type lookups need 3 \textit{ufnext} pointers to be dereferenced and more than 65% hit the representative itself.

**Graphs \( RV_{uf} \)** Figures 7.18, 7.19 and 7.20 show the graphs of \( RV_{uf}(i) \) as functions of the number of visited branches respectively. According to the graphs, \( RV_{uf}(1) \) has a tendency to decrease with the growth of the number of visited branches, while \( RV_{uf}(2) \) — to increase, just like \( RV_c \). Further experiments show that the tendency remains the same with the growth of the number of visited branches, though the rate of decrease (or increase) slows down continuously. Again the same argument as for \( RV_c \) explains the tendency.

**Conclusion** From the tests follows, that in more than 60% of the type or child lookups the passive merger is as efficient as the ideal active merger. In about 30% it is slightly less efficient — for type lookup it means two \textit{ufnext} pointer dereferences, for child lookup — two hash table accesses. Considering the ease with which passive merger handles the node merger and merger undo I think the tradeoff is worth it.
Figure 7.18: $RV_{uf}(1)$

Figure 7.19: $RV_{uf}(2)$
7.3.3 Conclusion: Average Cost of Type/Child Lookup in the Presence of Backtracking

Type Lookup
The average cost of a type lookup is \( R \times L + T \), where:

1. \( R \) is the average cost of finding the representative:
   
   \[
   R = \sum_{i=1}^{\max(R_{uf})} i \times \text{average}(RV_{uf}(i)) = 1 + \sum_{i=1}^{4} i \times \text{average}(RV_{uf}(i)) = 1 \times 0.775624 + 2 \times 0.216605 + 3 \times 0.007770 + 4 \times 0.000001 = 1.232148
   \]

2. \( L \) is the average cost of the link valid suffix removal:
   
   \[
   L = 1 + \sum_{i=0}^{\max(R_{isuf})} i \times \text{average}(RV_{isuf}(i)) = 1 + \sum_{i=0}^{3} i \times \text{average}(RV_{isuf}(i)) = 1 + 0 \times 0.998532 + 1 \times 0.001453 + 2 \times 0.000015 + 3 \times 0 = 1.001483
   \]

3. \( T \) is the average cost of the type valid suffix removal:
   
   \[
   T = 1 + \sum_{i=0}^{\max(R_{isuf})} i \times \text{average}(RV_{isuf}(i)) = 1 + \sum_{i=0}^{2} i \times \text{average}(RV_{isuf}(i)) = 1 + 0 \times 0.995950 + 1 \times 0.004000 + 2 \times 0.000050 = 1.004100
   \]
This gives us \(1.232148 \times 1.001483 + 1.004100 = 2.238075\), i.e. my implementation of backtracking makes a type lookup roughly twice as slower on average, then what it would have been without the backtracking handling.

**Child Lookup**

The average cost of a child lookup is \(C \times T \times L\), where:

\(C\) is the average number of individual nodes (contained in a compound node) needed to be visited in order to find \(\text{child}(N, \text{feat})\):

\[
C = \sum_{i=1}^{max(RV_c)} i \times \text{average}(RV_c(i)) = 1 + \sum_{i=1}^{8} i \times \text{average}(RV_c(i)) = 
\]

\[
= 1 \times 0.792065 + 2 \times 0.062031 + 3 \times 0.076931 + 4 \times 0.039548 + 5 \times 0.019702 + 6 \times 0.007988 + 
\]

\[
+ 7 \times 0.001677 + 8 \times 0.000058 = 1.463753
\]

This gives us \(1.463753 \times 1.004100 \times 1.001483 = 1.471933\), i.e. an average child lookup is slower by a factor of about 1.5 than the ideal average.

### 7.3.4 Composite Nodes Size/Dead Nodes Percentage

Recall from 3.3.2, that no nodes/arcs are ever deleted. A natural question is how many percents of all the nodes do the cancelled nodes constitute? While the previous statistics are collected during the unification, this one — after the unification has been done.

From the tests it appears that there are about 3\% of cancelled nodes on average, with maximum as many as 52\% (“181”, the total of 117 composite nodes).

**Definition 30** Composite node size.

Let \(N \in V_{cG}\), then \(|N|\) is called the size of the composite node \(N\). In other words, it is the number of individual nodes constituting the given composite node.

**Definition 31** Random variable \(RV_n\) denotes the size of a composite node.

The tests show that \(max(RV_n) = 32\), i.e. it can be quite large. However, such nodes are most likely to be leaves, since \(max(RV_c) = 8\) (see 7.3.2). Anyway, the chances to hit such a “large” node are very slim — see Table 7.14.
<table>
<thead>
<tr>
<th>i</th>
<th>Max $RV_n(i)$</th>
<th>Average $RV_n(i)$</th>
<th>the number of tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.000886</td>
<td>0.000002</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>0.000886</td>
<td>0.000002</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0.001129</td>
<td>0.000003</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>0.000472</td>
<td>0.000001</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>0.001168</td>
<td>0.000003</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>0.001222</td>
<td>0.000006</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>0.001193</td>
<td>0.000007</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>0.001395</td>
<td>0.000010</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>0.001771</td>
<td>0.000012</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>0.001486</td>
<td>0.000008</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>0.001372</td>
<td>0.000013</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>0.002336</td>
<td>0.000024</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>0.003667</td>
<td>0.000126</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>0.002083</td>
<td>0.000072</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>0.002370</td>
<td>0.000069</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>0.001414</td>
<td>0.000128</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>0.008299</td>
<td>0.000362</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>0.003886</td>
<td>0.000285</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>0.008380</td>
<td>0.000481</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>0.002099</td>
<td>0.000916</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 7.14: The largest values of $RV_n(i)$

In average 68% of the nodes appear to be of the unit size, 11% are composed of two individual nodes and 13% — of three (composite nodes consisting of 3 individual nodes occur more frequently than those of two).

### 7.4 Impact of Index

The :index annotation lets the unifier avoid some branches, which are a sure failure without actually entering them. While in LispFUF this technique allows for about 50% speedup, it has almost no effect on CFUF execution speed. Indexed features are usually the first ones in a branch, thus if the index check fails, then if we enter the branch it will be failed instantaneously. So, why is the speedup? The reason, is that entering a branch in LispFUF leads to creation of a new continuation (for backtracking) which is an expensive procedure. In CFUF, however, entering a branch costs almost nothing (because of the basic decision to optimize backtracking).

Still, there is an advantage in using indexing in CFUF. Recall, that the unification stops when the count of visited branches reaches some limit (see 6.1). Because a branch failed through the index check is not actually entered, this count is not incremented, thus enabling the unification to try
more branches. Consider the input “c35” (it comes with the original Lisp-
FUF package). During the unification of it with the SURGE grammar 2740
branches are visited when the index is disabled and 1996 — when enabled.
The current branch count limit is 2500 branches. As you can see indexing is
a matter of success and failure for this particular input.

There is, however, a second side to the medal. Let $fd$ be an input known
to fail regardless of whether indexing is used or not. Then it will take more
time for $fd$ to fail with indexing than without. The reason is precisely the
same — without indexing “useless” branches are visited, they fail immedi-
ately, but contribute to the branch count. With indexing — more complex
branches are entered instead, they take more time to complete than the use-
less ones. The result is that both ways $fd$ fails, but it can be twice as slow
with indexing.

7.5 Summary

- CFUF vs. LispFUF.
The comparison shows that CFUF performs significantly faster than
LispFUF and the larger the FD/grammar the larger the speedup. Speedup
on complex FDs can be as high as 100.

- CFUF profiling.
It can be concluded from the evaluation results, that CFUF is no more
than twice as slow as the ideal FUF implementation based on the trans-
lation of FD to a rooted directed graph and FDVM execution. That
is, the cost for handling backtracking is to multiply by two the basic
graph traversal operations.
Chapter 8

Implementation Issues

8.1 Overview

CFUF is written in C++ [12]. It is a complex system, consisting of 6 modules:

1. Base.
   Implements basic data structures (like type and arc) used by all the other modules. This is like the foundation of a building, all of the things to come depend on it.

2. Parser.
   This module is responsible for parsing FDs and grammars. FDs and grammars supplied by the user are lispish lists. This module checks the syntax and issues the appropriate compilation commands during the parsing.

3. Compiler.
   This module is responsible for compiling FDs and Grammars. It builds the internal representations of FDs and grammars. In effect, it implements the FDVM (see 3.4).

4. Unifier.
   This module performs the unification of compiled FDs and grammars with the help of the FDVM, from the compiler module.
5. Tracer.
   This module will allow the user to monitor the unification process. Its capabilities are rather limited right now.

6. Interpreter.
   Accepts input from the user.

The first 5 modules form the **CFUF core**. The core does not depend on the interpreter, though being detached from the user it is pretty useless.

### 8.2 Interpreter

Unlike the interpreter used in the original FUF implementation (Common Lisp), CFUF comes as an inferior package to SCM — the scheme language interpreter implemented by Aubrey Jaffer (http://swiss-ftp.ai.mit.edu/~jaffer/index.html). The interpreter module provides the necessary minimum of routines, called primitives, enough to port existing applications written in LispFUF. The debugging support is still under development, only constituents and branches can be traced currently. Interactive debugging is absent altogether, though there is a way to manually define the order in which constituents are enqueued (see 6.3).

As I have already mentioned, the CFUF core does not depend on the interpreter used in the interpreter module, as long as it obeys certain protocol. The core assumes that there exists an interpreter object derived from the `CInterp` protocol class defined in the base module. When the core is initialized it is passed that pointer. The protocol defines the way to evaluate the `control` and `extern` statements.

To change interpreter should be relatively easy. All is needed is to replace the interpreter module with another one. This includes three major steps:

1. The interpreter class must be derived from `CInterp`.

2. The parser module operates on tokens supplied by the lexer, which in turn reads symbols from the object of lexer stream class derived from `CInputSteam` protocol class. The new interpreter module may be required to implement this lexer stream class. If the new interpreter is Common Lisp, then the lexer stream class for SCM can be used.
3. The primitives must be reimplemented. This should not be hard, since the primitives do not change, only the interface with the interpreter language changes.

8.3 Memory Management

The CFUF core manages the memory by itself. The type objects, being the most "mobile" objects in the system, are used in the form of smart pointers implementing the reference counting technique. Most of the ideas for the memory management were taken from the Scott Meyers' books "Effective C++" and "More Effective C++" [10], [9].

The objects created in the interpreter, including compiled FDs and grammars, are managed by the SCM Garbage Collector. The SCM code is not aware of the memory used to represent FDs and grammars by the core, thus it fails to call GC automatically, when there are too many compiled objects. This can be fixed by tracking the memory used by each compiled FD.

Right now the GC is called manually from within the interpreter module each time a certain internal variable reaches some threshold. All the compiled objects have a GC weight associated with them. For example, the grammar weight is 2, the weight of an FD coming from the unification is 3, the weight of a timer object is 0.1 (timers are used to measure execution time). Every time such an object is created, the variable is incremented by the object GC weight. The user cannot vary the GC weights and threshold, but that capability can be easily granted.

8.4 Containers

The role of containers in every large system is hard to underestimate. It is crucial that they be implemented as efficiently as possible.

Instead of inventing the bicycle, I use the Standard Template Library (STL) developed by Alex Stepanov and Meng Lee. There are a number of implementations of STL, I prefer the one from www.sgi.com/Technology/STL.
Chapter 9

Conclusion

This work has presented CFUF — a new, efficient implementation of an interpreter for the FUF language.

CFUF is fully compatible with the existing implementation of LispFUF that has been available since 1990. It can run unchanged the SURGE generation grammar of English which has been developed in FUF.

CFUF is built on the model of a virtual machine for the execution of FUF programs. FDs are translated into an internal graph representation. The set of commands to manipulate this graph representation (FDC) is the base language of the FDVM (FD Virtual Machine).

FUF grammars are translated into an AND–OR tree where each and–node contains a sequence of FDC commands. The FUF interpreter traverses the AND–OR tree in a non–deterministic manner and executes the FDC commands in the right context as encoded in the set of registers of the FDVM.

The design of the FDVM takes into account the empirical fact that FUF programs are extremely non–deterministic and include lots of backtracking. We have therefore favored in the design decisions fast handling of backtracking and undoing. Particular attention has been paid to avoid copying structures in memory as well, which is known to cost a lot of the practical runtime of unification–based formalisms. The result is a design based on a “clock mechanism” where modifications to the FD graph are never undone, but simply made invalid when a clock is advanced. Traversal of the graph is made, as a result more expensive, but the tradeoff is shown to be beneficial.

Detailed performance evaluation is provided that demonstrates the well–foundedness of the major design decisions. When compared with LispFUF,
on large inputs, CFUF proves to be close to 100 times faster depending on the benchmark. It is shown that the larger the input, the better the speedup — that is, empirically, the order of growth of the CFUF interpreter is better than that of LispFUF.

A detailed profiling analysis and execution model demonstrates that the clock-based model of FD brings minimal overhead during the basic operations of the graph traversal (type queries and traversal from node to child node). In return for this overhead, backtracking is almost free. This feature of the CFUF design is made clear in the analysis of the impact of the Index special annotation. In LispFUF, index can improve unification time by as much as 50%. In CFUF, the impact of index is extremely small. This is because index is basically used to minimize backtracking that would have failed immediately anyway.

The practical performance of CFUF makes it possible to use large-scale FUF-based grammars such as SURGE for real-time generation applications. Memory usage is minimal (a footprint of 16Mb of RAM is sufficient). CFUF is integrated with a Scheme interpreter to facilitate interactive experimentation.

In the future, the need to develop a visual interface to FUF for grammar development and grammar tracing and debugging becomes more important. The development of CFUF has made it possible to envision a usable development platform that can be entirely based on a fast, compact unification engine.
Bibliography


