Machine Translation and Type Theory

Aarne Ranta

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

To Per Martin-Löf.

Abstract. This paper gives an introduction to automatic translation via examples from the history of the field, where statistical and grammar-based methods alternate. Grammatical Frameword (GF) is introduced as an approach that uses type theory to provide high-quality translation between multiple languages. GF translation is fundamentally grammar-based but can be combined with statistical methods such as learning translation models from a corpus and ranking translation candidates by probabilities. [1]

Keywords: machine translation, type theory, Grammatical Framework, multilingual grammar

1 Introduction

Machine translation was one of the first applications envisaged for digital computers. The need came from the U.S. military: computers would help intelligence by automatically translating Russian documents to English. This enterprise was encouraged by the success of cryptography during the Second World War. Russian was seen as an encrypted form of English, and translation was a matter of cracking the code.

The main ideas were summarized by Weaver in 1947 (Hutchins 2000). He proposed several approaches, but the most influential one was the use of the noisy channel model developed by Shannon (1948). In practice, most of the early work had to do with word-to-word translation—how to store large dictionaries and perform efficient lookup with the machines of the time. But Weaver and Shannon also envisaged the generalization of this model to *n-grams* of words. The rationale was that single words are often ambiguous and have many alternative translations, but they can be disambiguated in a context.

For example, *even* in English has French translations such as *même* (adverb), *égal*, *plat*, *pair* (adjectives). In the 2-gram *even number*, *pair* is the likely translation, whereas *not even* might become *même pas*, as in *he does not even smile*. However, in the sentence *7 is not even*, the adjective *pair* is again the right choice, which might be detected if 4-grams are considered, and so on.

It is easy to find counterexamples to n-gram based translation for any fixed n. And even for relatively small values of n, the method has scalability problems. A natural

[1] Preprint of a paper appeared in *Epistemology versus Ontology. Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*. Editors: P. Dybjer, Sten Lindström, Erik Palmgren, G. Sundholm. Logic, Epistemology, and the Unity of Science Volume 27, Springer-Verlag, 2012, pp 281-311.

language may have millions of words, if all word forms are counted separately, as customary in approaches that don't use grammars. The number of n-grams is exponential in n, and *sparse data* becomes a problem. If the n-gram model is based on a corpus of text, there is no hope to find all relevant n-grams. Thus the actual translation usually has to recur to smoothing with smaller n-grams, which results in lower quality.

The problems with statistical n-gram based translation were of course known to their developers. They did not expect the translation to give more than approximations, or *raw translations*, which were to be improved by human post-processing. Alternative methods were studied in parallel. Thus Bar-Hillel soon published the idea to use *categorial grammars* for translation (Bar-Hillel 1953). This was an elaboration of the idea of Ajdukiewicz (1935) to use simple type theory to formalize the rules of natural language. The advantage of grammar rules compared to n-grams is that they can easily cope with arbitrarily long sequences. Thus for instance, in *this number was always believed but never proven to be even*, there are 8 words between *number* and *even*, but these words are related by grammar, which is easy to describe by Bar-Hillel's model. More generally, since a grammar can cover an unlimited number of sentences, it can overcome the problem of sparse data.

Bar-Hillel devised an algorithm for what was later to be known as context-free parsing. Much of his work was not possible to implement on the computers of the time and remained theoretical, although seminal. But his most famous contribution to the field was his eventual rejection of the whole enterprise of machine translation (Bar-Hillel 1964). He showed with some simple examples that the translation problem may require unlimited intelligence and universal knowledge. His examples used the word *pen*, which can mean either a writing utensil or a play area for children. In most languages, these two senses of *pen* have two different words. Now, the two sentences *the pen is in the box* and *the box is in the pen* probably use *pen* in these two different senses. But how do we know? The knowledge does not come from grammar, but from our familiarity with the sizes of objects in the world. Bar-Hillel concluded that fully automatic high-quality translation is impossible, not only in foreseeable future but in principle, because there is no hope to formalize all this knowledge.

After Bar-Hillel's paper, the ALPAC report (Automatic Language Processing Advisory Committee) was published in 1966 (Pierce & al. 1966). It presented an evaluation of the results obtained with all the massive funding given to machine translation in the post-war era. Its drastic conclusion was that the investment was wasted—that machine translation had not delivered anything useful. The consequence of the ALPAC report was that machine translation funding in the U.S. was withdrawn and the projects laid down.

Interestingly, the main argument in the ALPAC report was not that machine translation was bound to be unreliable, but that it was too expensive. It was not denied that machine translation *can* give good results, but the problem was that, to achieve these results, so much work was needed that manual translation was cheaper. The explanation pointed out was that machine translation had been pursued as pure engineering task in an *ad hoc* manner—that one should first have done scientific groundwork and described the languages before attacking the complex engineering problem of machine translation. And indeed, what replaced machine translation as the occupation of many

groups and individuals was the field of *computational linguistics*, which now emerged with more modest and realistic goals than machine translation.

In the 1970's and 1980's, most of computational linguistics focused on creating mathematical language models and collecting data, rather than building ambitious machine translation systems. Many of the exceptions had to do with more modest goals than full-scale machine translation: thus *domain-specific translation systems* were able to build accurate models for limited areas such as weather reports (Chandioux 1976). On such limited areas, one can often avoid the problem with word-sense ambiguity; for instance, if the area is elementary arithmetic, then *even* can be assumed to mean divisibility by 2. More generally, the semantics of a limited domain can be formalized in such a way that meaning preservation becomes a rigorously defined concept.

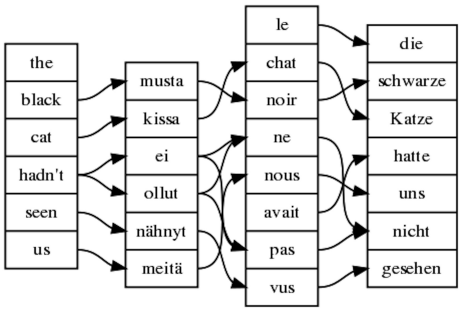
Another idea that emerged after the ALPAC report was to make translation *interactive*. In 1979, Kay wrote an influential paper (eventually published as Kay 1997) that showed with more examples in the spirit of Bar-Hillel that proper translation needs human judgement. Many of Kay's examples had to do with the interpretation of pronouns. For instance, *il* in French can be translated by both *he* and *it* in English. Thus *il est possible que...* becomes *it is possible that...*, whereas *il est convaincu que...* is more likely to become *he is convinced that...*. Translating the pronoun is, just like lexical disambiguation in Bar-Hillel's case, the matter of an unpredictable amount of information about the context of use and the world.

We will return to the problem of translating pronouns in Section 4. Kay's main suggestion was that, since there is no automatic solution, translation tools should make human intervention seamless—must more seamless than post-processing. This idea indeed led to practicals tool widely used by professional translators—basically text editors in which translation is performed manually. Such tools are equipped with a *translation memory*—a repository of sentences and phrases translated earlier, reducing the need of translating the same phrase more than once.

The post-ALPAC pessimism around machine translation was finally removed at the end of the 1980's, in the project funded by IBM and led by Jelinek (Brown & al. 1990). The "IBM approach" was another try with Shannon's noisy channel model and n-grams, but now with the benefit of more powerful computers, adequate bilingual training data (from the Canadian parliament), and experience from statistical speech recognition likewise conducted by Jelinek a decade earlier (Jelinek 2009). A mathematician and engineer himself, Jelinek was disappointed by the over-sophisticated and complex models of linguists, which required so much human effort and gave so little concrete results. The IBM approach created with relatively little effort a system that performed fully automatic translation of unlimited text.

The quality reached by the IBM approach was not perfect, but it was continuously improved, and methods were developed for building translation systems with minimal human intervention. The most prominent achievement of this approach is Google's online translation system (`translate.google.com`), which at the time of writing covers 57 languages. Google translate is *robust* in the sense that it produces a result for any input. Moreover, it has been created with a minimal human effort, in particular, with no or little linguistic knowledge about the languages involved. Instead of linguistics, it uses techniques such as *phrase alignment* (Och and Ney 2004), which extract combi-

nations of words ("phrases") and their translations from parallel texts. Actually, it has become a rule of the game that no linguistic, language-specific knowledge may be used, but translation systems must be built by language-neutral algorithms. This can require a lot of data, because alignment can be a many-to-many relation and work on a long distance. The following picture shows a set of alignments between English, Finnish, French, and German.



One of Google's obvious strengths in machine translation is that the company has access to so much text, perhaps more than any other organization in the world. In this field, "there is no data like more data". Hence, if anyone is able to solve problems by introducing more data, then Google should be. Nevertheless, the results are not satisfactory in all respects. Google translate is useful in giving quick rough translations, and impressive in its speed of increasing the number of languages. But its quality cannot be trusted, and it is doubtful whether it can ever become fully reliable. This is due to the general reasons given by Bar-Hillel and Kay, but also to the specific limitations of purely statistical language models. To remove the latter kind of problems, much of the current research in machine translation targets *hybrid models*, which combine statistical processing with linguistic information (Lopez 2008 gives an excellent survey of such methods).

2 Models of Translation

The history of machine translation shows a back and forth movement between statistical and ruled-based methods. In its purest form, statistical translation is Shannon's noisy channel decoding based on n-gram probabilities. Rule-based translation, on the other hand, applies manually written translation functions and performs disambiguation by deep semantic analysis. This model is actually the same as is used in *compilers* for programming languages.

A compiler is a translator from a source language, such as C or Java, to a target language, such as the Intel x86 machine language or JVM (Java Virtual Machine). In early times, compilers were implemented as *transducers* directly converting source code to target code. This was accomplished by means of *semantic actions* attached to the grammar rules of the source language; the mathematical model of this was given in the *attribute grammars* of Knuth (1968). For example, the rule for addition expressions could look as follows:

```
Exp ::= Exp "+" Exp
  { compile $1 ; compile $3 ; emit (ADD (type $1 $3)) }
```

This rule implements infix expressions with the operator +. It compiles the first operand `$1` (emitting whatever code belongs to it), recognizes the + sign (which would be denoted by `$2`), compiles the second operand `$3`, and finally emits the `ADD` instruction computed from the types of the operands by the attribute `type`. This attribute makes it possible to use an *overloaded* addition operator in the source language, at the same time as the machine language typically has separate addition instructions for integers, floating point numbers, etc. Notice that the resolution of overloading is similar to word sense disambiguation in natural language translators: the source language can have words such as *even* in English, which require an inspection of the operands (e.g. the nound modified) to decide about the correct translation in a target language like French.

Semantic actions in transducing compilers can be very complex, since they simultaneously define several different operations, such as type checking and code generation in the example above. Furthermore, one-pass transduction poses serious constraints on the source language, which has to be closer to machine languages and thus less natural for humans than modern high-level programming languages are. For instance, mutual recursion is difficult to deal with in one-pass compilers.

Modern compilers thus favour several passes, most of which operate on an *abstract syntax*, which is an intermediate representation between the source and the target language (Appel 1998). The abstract syntax can be formalized as a system of datatypes, where the data objects are abstract syntax trees. The first phase of the compiler is *parsing*, which converts the source code string into an abstract syntax tree. The last phase is *linearization*, which converts abstract syntax into target code. Between these phases, several operations of code analysis and optimization can be performed to manipulate the abstract syntax tree. For instance, GCC (the GNU Compiler Collection, Stallman 2001) can make dozens of passes before emitting the target code.

In addition to modularizing the compiler, the use of an abstract syntax makes it language-neutral: it can be applied to new source and target languages by just changing the parsing and linearization components. The hard work (semantic analysis and optimizations) is performed on the abstract syntax level. Thus GCC, which was originally created for translating C into Motorola 68020, currently supports several source and target languages.

The two compilation methods discussed above have counterparts in the translation of natural language. The transduction model corresponds to *transfer*, i.e. translation functions defined separately for each pair of languages. The abstract syntax model corresponds to *interlingua*. Just like in compilers, the interlingua is an abstract representation of meaning, and translation is performed by meaning-preserving mappings between the interlingua and the languages involved. Thus the translation from English to French is the composition of first translating English to the interlingua and then the interlingua to French.

The advantages of the interlingua approach are the same in machine translation as in compilers. A well-designed, semantically grounded interlingua is an excellent platform for the analysis of the source language, and tasks such as word sense disambiguation and anaphora resolution. It is also useful when selecting the most natural expressions

in the target language—an operation similar to *optimizations* in the case of compilers. Another advantage is similar to the multi-source multi-target compilers: work is saved, both in the semantic operations (which are language-independent) and in the number of translation functions. An interlingual system involving n languages needs just $2n$ functions: from each language to the interlingua and back. If separate transfer functions were used for each pair of languages, $n(n\text{-}1)$ functions would be needed for n languages.

The transfer/interlingua distinction is orthogonal to the statistical/rule-based distinction. In both types of translation, it is the interlingua approach that scales up into highly multilingual systems. Thus Google translate uses an interlingua for most of its 57×56 language pairs. This interlingua is English (as confirmed by Franz Och in personal communication).

From the semantic point of view, English (or any natural language), might sound like a strange interlingua, because it is ambiguous and destroys distinctions found in other languages. To take a typical example, the distinction between singular and plural *you* disappears in English. Consequently, the translation between, for instance, Swedish and French is not guaranteed to preserve this distinction. Swedish *jag älskar dig* ("I love you" (singular/familiar)) and *jag älskar er* ("I love you" (plural/polite)) are currently both translated as *je t'aime* ("I love you" (singular/familiar)), although the plural/polite form should be translated *je vous aime*.

Nevertheless, English is probably the best choice for training statistical translation models, because there is much more data available for Swedish and English in parallel and for French and English in parallel than for Swedish and French in parallel. There is, moreover, a compelling reason for using a natural language as an interlingua: there simply is no formal language capable of expressing everything that can be expressed in natural languages. This requirement for an interlingua was formulated already by Descartes in 1629, when he proposed a universal language that would support translation:

> [the universal language must] establish an order among all thoughts that can enter in the human spirit, in the same way as there is a natural order among numbers, and as one can learn in one day the names of all numbers up to infinity and write them in an unknown language, even though they are an infinity of different words...
> The invention of this language depends on the true philosophy; for it is impossible otherwise to denumerate all thoughts of men and order them, or even distinguish them into clear and simple ones...
> (Descartes, letter to Mersenne 1629)

The need of such precision, of a "true philosophy", is demonstrated by the examples of Bar-Hillel and Kay: if the interlingua were to determine how to express the meaning of the source in the target language, it has to be unambiguous and make all the required distinctions. Now, as centuries of philosophers and logicians have in vain been looking for such a formalism, shouldn't we admit that it is just an unrealistic dream?

A natural idea is to use logic and type theory when building an interlingua. An early proposal to this effect was made by Curry (1961). It was applied at a larger scale in the Rosetta system (Rosetta 1994) at Philips. Rosetta was based on the grammar and logical semantics of Montague (1974). They were generalized from English to a multilingual

grammar in a way that contained many ingredients of the method discussed in Section 6.

3 A Framework for Translation

The previous discussion has identified two distinctions within machine translation:

- statistical vs. rule-based
- transfer vs. interlingua

We will now propose a framework for translation, which is rule-based and uses an interlingua. But we will later show how this model can be extended with statistical components and transfer. We will also meet the main challenge of the ALPAC report and show that the framework is economically viable and useful.

The framework has the same structure as multi-source multi-target compilers: a translator consists of an abstract syntax together with mappings to and from concrete languages. The concrete languages can be varied *ad libitum*; the technique should apply to all natural languages. But it can also deal with formal languages, in tasks such as translating between predicate logic and English.

While having the same structure as GCC, a translation framework must be more general, so that it can deal with different subject matters and not only with computer programs. Thus it must have a more expressive abstract syntax than GCC. It might seem that we would need the power of a universal interlingua—but fortunately we don't. Instead, we apply the idea of a *logical framework* (LF, Harper & al. 1993), originally designed to be a *framework for defining logics*, as a *framework for defining interlinguas*. Then we can define *domain-specific* interlinguas, corresponding to semantically coherent and formalizable domains. To coin a slogan, *the Rosetta stone is not a monolith but a boulder field*.

Logical frameworks were born in the constructivist tradition, which abandoned the idea of a monolithic foundation of mathematics. Instead of reducing all mathematics into one formal theory, such as axiomatic set theory, a logical framework makes it possible to define separate theories for different parts of mathematics. With the expressive power of *dependent types*, this extends to the possibility to define new systems of inference rules, that is, new logics. The framework itself doesn't determine a logic, but provides an infrastructure with a common notation, algorithms for proof checking and proof search, and a generic user interface. Ever since the early times of LEGO (Luo and Pollack 1992), Coq (Dowek & al. 1993), and ALF (Magnusson 1994), logical frameworks have provided an economical way to implement logics and experiment with them. Due to the infrastructure provided by the framework, the implementation of a new logic boils down to writing a set of declarative definitions.

The logical frameworks LEGO and ALF were based on the constructive type theory of Martin-Löf (Martin-Löf 1984, Nordström & al. 1990). Constructive type theory has also proven usable for meaning representation in natural languages (Ranta 1994). The type checking and proof search machinery provided by a logical framework gives tools for the kind of semantic analysis needed in machine translation. What is missing are the parsing and linearization functions for the natural languages themselves. To fulfill this

need, the *Grammatical Framework*, GF (Ranta 2004, 2011), was developed. GF is an extension of a logical framework with a component called *concrete syntax*.

If LF is a framework for defining logics, GF is a framework for defining *multilingual grammars*. A multilingual grammar is a pair

$$< \mathcal{A}, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\} >$$

where $\mathcal{A}$ is an abstract syntax (a logic in the sense of LF) and $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are concrete syntaxes. A concrete syntax is a mapping between the abstract syntax trees of $\mathcal{A}$ and the strings in some string language, such as English, French, Java, or JVM.

As a first example of multilingual grammars in GF, consider the translation of addition expressions. The abstract syntax defines a *function* (`fun`), and each concrete syntax defines a *linearization* (`lin`). The following grammar covers Java, JVM, English, and French.

```
fun EPlus : Exp -> Exp -> Exp
lin EPlus x y = x ++ "+" ++ y
lin EPlus x y = x ++ y ++ "iadd"
lin EPlus x y = "the sum of" ++ x ++ "and" ++ y
lin EPlus x y = "la somme de" ++ x ++ "et de" ++ y
```

The `lin` rules of GF are *reversible mappings*: they can be used both for the linearization of trees into strings and for the parsing of strings into trees. How to do linearization is obvious: just think of the `lin` rules as clauses in the definition of a recursive function, where the variables `x` and `y` stand for the linearizations of the arguments. The parsing direction is more tricky and can be stated as a search problem. A general solution was found by Ljunglöf (2004), who moreover showed that the parsing complexity in GF is polynomial.

The above example is a valid GF grammar, but it is oversimplified in many ways. First we might notice that the abstract syntax `fun` rule doesn't indicate the type of the expression (integer, float, etc). The JVM rule is, in an arbitrary way, directed to integer addition (`iadd`) only. But this problem can be solved by making `Exp` into a *dependent type*, which takes the object language type (type `Typ` in this grammar) as its argument. Then we can write

```
fun EPlus : (t : Typ) -> Exp t -> Exp t -> Exp t
```

to force the operands and the value to be of the same type, and

```
lin EPlus t x y = x ++ y ++ add t
```

to select the proper JVM instruction `add t` as a function of the type `t`. (Precisely how the `add` function is defined in GF is omitted here.) In the other three languages, the type argument is *suppressed*. For instance,

```
lin EPlus _ x y = x ++ "+" ++ y
```

We use the wildcard _ for arguments that are suppressed, that is, don't appear on the right of the equality sign. Now assume the following rules for numeric literals and program variables, with linearizations in Java:

```
fun EInt : Int -> Exp TInt
lin EInt i = i
fun EVar : (t : Typ) -> Var t -> Exp t
lin EVar _ v = v
```

In JVM, the type of the variable has to be known by the instruction that loads the values of variables from memory:

```
lin EVar t v = load t v
```

As Java suppresses the type arguments of `EPlus` and `EVar`, the expression `2 + x` is initially parsed by introducing *metavariables*:

```
EPlus ?1 (EInt 2) (EVar ?2 x)
```

If `x` is an integer variable, well-known algorithms for type checking and constraint solving, similar to those used in ALF (Magnusson 1994), now manage to instantiate the metavariables:

```
EPlus TInt (EInt 2) (EVar TInt x)
```

From this syntax tree, we can generate the JVM code

```
iconst_2
iload_0
iadd
```

which uses the `i` (integer) variants of the `load` and `add` instructions. (It moreover maps the variable `x` to the memory address 0, but we omit the details about how this is done.)

4 Types and Disambiguation

Although the translation of `2 + x` to JVM is elementary, it illustrates some fundamental aspects of machine translation:

- word order can vary from one language to another (here: infix in Java, postfix in JVM)
- one language may suppress information that another language needs (here: the type of the addition operator)
- suppressed information can be restored by semantic analysis (here: type checking and metavariable solving)

A natural language example with the same features is *anaphora resolution*, that is, the interpretation of pronouns. Consider the following examples (from Hutchins & Somers 1992):

the monkey ate the banana because it was hungry
the monkey ate the banana because it was ripe
the monkey ate the banana because it was tea-time

The focal point is the pronoun *it*. The proper translation into German is different in each of the three sentences. In the first one, *it* refers to the monkey (*der Affe*), and becomes the masculine *er*. In the second one, *it* refers to the banana (*die Banane*), and becomes the feminine *sie*. In the third one, *it* is the formal, impersonal subject, translated by the neutrum pronoun *es*.

What is the algorithm for translating *it* in the three described ways? As is clear from the explanations given to each translation, it has to do with the *reference* of the pronoun, not just its syntactic form. It also has to do with the *type* of applicability of the adjective that it predicated of the pronoun. The outline of the algorithm presented in Ranta (1994) is the following:

1. Analyse the context of the pronoun to collect all possible referents with their types, thus forming the *referent space* $\{r_1 : R_1, \ldots, r_n : R_n\}$ of objects given in the context.
2. Analyse the occurrence of the pronoun and collect all types $\{T_1, \ldots, T_m\}$ that an object may have in that position.
3. Consider the set of those elements $r_i : R_i$ whose type R_i matches some of the types T_j.
   (a) If the set is singleton $\{r_i : R_i\}$, then r_i is the referent and its type is R_i.
   (b) If the set is empty, then report an anaphora resolution error (or widen the referent space).
   (c) If the set has many elements, then ask the user to disambiguate interactively (or look for more constraints in the context).

This algorithm does a half of the job—it finds the referent of the pronoun with its type. The other half is to generate the translation. But this part is easy once the referent and its type are found, because pronouns can be given the abstract syntax

```
fun Pron : (t : Typ) -> Ref t -> Exp t
```

and the German concrete syntax chooses the proper word as a function of the gender of the type,

```
lin Pron t _ = case (gender t) of {
  Masc  => "er" ;
  Fem   => "sie" ;
  Neutr => "es"
  }
```

(showing only the nominative forms for simplicity). When the parser encounters the English pronoun *it*, the initial abstract syntax tree is

```
Pron ?1 ?2
```

But as soon as the resolution algorithm has found a value for `?1`, the translation can be performed.

The above algorithm is a sketch, as it uses undefined concepts and leaves alternatives open. First, we need to know how to "analyse the context". We use *context* in the technical sense of type theory: the sequence of variables with their types that are in scope. This context is maintained by the type checker when it analyses the syntax tree. The *possible referents*, then, are a closure of the context under simple operations such as the projections p and q of Sigma types. Ranta (1994) explains in more detail how different constructs of natural language contribute to the context.

Secondly, what are the types that "an object may have at the position" where the pronoun occurs? This can be defined by considering the wider syntax tree around the pronoun. In general, there can be many such trees because natural language is syntactically and lexically ambiguous. These trees have the form $t_i(x)$, where x is the slot for the pronoun. The types $T_i : \{T_1, \ldots, T_m\}$ are thus all the types that x can have in all the trees $t_i(x)$. In addition, the pronoun of the type T_i must be the one actually being resolved (i.e. have the same gender); this is the only condition referring to concrete syntax in the algorithm, which otherwise works on the abstract syntax level. (Notice that we have here assumed that the number of types is finite; if this doesn't hold, the problem may become undecidable.)

The third concept left undefined is *type matching*. The baseline is equality: $R_i = T_j$. But this can be extended by using techniques such as *coercive subtyping* (Luo and Callaghan 1999). Johannisson (2005) and Angelov and Enache (2010) show how to do this in GF. Considering the first of the examples above, the predicate *hungry* might be defined as a propositional function over animals,

```
fun Hungry : Exp Animal -> Prop
```

The monkey, on the other hand, might be introduced as a referent of type `Monkey`,

```
r : Ref Monkey
```

But a coercion

```
c : Exp Monkey -> Exp Animal
```

will establish

```
c (Pron Monkey r)
```

as a possible argument of `Hungry`. Notice that we use the coercion on the `Exp` level rather than `Ref`, so that the gender of the subtype is preserved.

Fourthly, what does it mean to *widen the search space* when no referent is found? One way is to widen the class of the operations under which the referent space is closed. Subtyping coercions can be seen as an instance of this, but any functions may have to be considered. This shows that anaphora resolution can be as hard as *proof search* in general. Another way is to widen the context that generates the search space. For

instance, if the referent cannot be found in the same sentence as the pronoun, earlier sentences may have to be taken into account. One reason for the undecidability found by Bar-Hillel (1964) and Kay (1997) is that the search space may be infinite.

Fifthly, what does it mean to *look for more constraints* when the referent is not unique? One possibility is to shrink the search space. For instance, if the context has been widened by earlier sentences, priority can still be given to referents found in later ones. Another possibility is to use probabilities. Consider, for instance,

the monkey ate the banana because it was so sweet
the monkey ate the banana because it had fallen from the tree

The above algorithm may construct both the monkey and the banana as possible arguments, but one of them may be more likely, and this can be defined by using *probabilistic GF grammars* (see Section 8 below). However, in the end maybe none of the referents comes out as the clear winner, or the translation task may be so critical that no guesses are tolerated. It is in these cases that the system may need user input and hence be interactive. In a good translation system, interactive disambiguation should be smooth and intuitive. The systems should, for instance, not display types or abstract syntax trees, but rather pose simple questions in natural language, in a manner similar to how a human would do: *do you mean the monkey or the banana?*

In its full generality, type-theoretical anaphora resolution is undecidable—just as the arguments of Kay (1997) suggest. The translation of large documents may preclude the use of interaction. But the algorithm can still be seen as the *specification* of what anaphora resolution should ideally do. Practical approximations can be created by omitting too complex proof search or far-away parts of the context.

Also statistical models can give approximations of anaphora resolution: since all words in *it was hungry* fit into one 3-gram, it may well happen that a model "knows" that *it* is related to *hungry* and guesses the translation of the pronoun right.

5 User Interaction

As fully reliable translation cannot be fully automatic, user interfaces are an essential part of machine translation. Post-processing bad machine translation is hardly a sufficient form of interaction; one of the conclusions of the ALPAC report (Pierce & al. 1966) was that translators found it slow and unpleasant, and would have preferred manual translation from the beginning. In grammar-based translation, grammars that are accurate enough for translation can hardly be complete. Hence their users easily end up in situations where the input is not recognized, and the response from the system is "syntax error". While this is accepted in compilers, where the grammars can be learnt from manuals, it is hardly acceptable in parsers of natural language, where grammars are theoretical constructs not known by native speakers.

The model that has been applied in the user interfaces of GF is that of a *syntax editor* (Teitelbaum and Reps 1981, Donzeau-Gouge & al. 1975). Syntax editors have been a standard interface for logical frameworks, where they are backed by a rich metatheory of editing actions (Magnusson 1994, Norell 2007). The main idea of a syntax editor is that the user is manipulating abstract syntax trees and not texts; the text is just a special

view of the tree, produced by linearization. One advantage of syntax editors is that they avoid the problem of parsing. An early application of this was the WYSIWYM system ("What You See Is What You Mean", Power and Scott 1998), which replaced translation by *multilingual generation*. The user of WYSIWYM would directly construct an abstract representation, from which translations in different languages were generated automatically.

Pure syntax editing can however be heavy and slow, and it requires the awareness of an abstract, sometimes complex, structure. When comparing parsers and syntax editors, Welsh & al. (1991) ended up recommending *pluralistic editors*, which combine parsing and syntax-based editing. Now, since parsing has been supported for GF grammars from the beginning, the syntax editors built for GF have been pluralistic (Khegai & al. 2003). Their basic functionality is the stepwise construction of syntax trees by *refinements*, which are selections of constructors that build trees in a top-down fashion. For example, the construction of an arithmetic expression can have as its intermediate state the tree

```
EPlus ?1 ?2
```

In this state, a refinement is expected for the first metavariable, `?1`. This refinement can be selected from a *menu*, which contains all constants and variables whose value type is possible for `?1`. In this case, the menu might contain the constructors `EInt`, `EVar`, and `EPlus`. If `EPlus` is selected, the next state is

```
EPlus (EPlus ?11 ?12) ?2
```

However, as the editor is pluralistic, it also accepts an expression written in a concrete syntax as a refinement. Refining `?1` by `the sum of x and 5` would thus result in

```
EPlus (EPlus (EVar x) (EInt 5)) ?2
```

compressing five refinement steps into one short string. Since the editor continuously type-checks the tree, it of course makes sure that the variable `x` is actually available in context and has a correct type.

The disadvantage of parsing compared with syntax editing is that syntax errors are possible. Fortunately, a pluralistic editor can solve this issue by *incremental parsing*—a process in which the input is analysed word by word, and the set of possible next words is computed after each word. Thus a user may start typing

```
every number is _
```

and get a list of suggestions: *divisible*, *equal*, *even*, *not*, *odd*, *prime*, etc. Every suggestion is guaranteed to lead, eventually, to a correct sentence and thereby an abstract syntax tree. If the list of suggestions is long, it can be narrowed down by typing the beginning of a word: with

```
every number is e_
```

only words beginning with an *e* are suggested. If the incremental parsing algorithm is efficient and the interface well implemented, its usage can be as fast as the input of free text. It can even be faster, because typos are excluded and unique word choices can be auto-completed. Angelov (2009) defines the incremental parsing algorithm actually used in GF. A later version of the algorithm (used in Angelov and Enache 2010) integrates dependent type checking and variable binding analysis to narrow down the suggestions to semantically correct ones.

For most users, incremental parsing is the method of choice when source text is created in the first place. However, syntax editing can still be useful in later edits of a text. Consider the following business letter written in French:

> *Chère Madame X, j'ai l'honneur de vous informer que vous avez été promue chargée de projet.*

("Dear Mrs X, I have the honour to inform you that you have been promoted to a project manager"). If Mrs X declines and the letter is sent to Mr Y instead, just changing the recipient will result in

> ***Chère** Monsieur Y, j'ai l'honneur de vous informer que vous avez été **promue** **chargée** de projet.*

The boldface parts of the letter are now grammatically incorrect, since they are in feminine forms, in agreement with *Madame X*; embarrassingly, they may disclose to *Monsieur Y* that he was not the first choice for the position. But this embarrassment can be avoided if the letter is constructed in a syntax editor and the abstract syntax tree is saved. If the tree has the form

```
Letter (Dear (Mrs X)) (Honour (Promote ProjectManager))
```

then the one-place change of `Mrs X` to `Mr Y` results in

```
Letter (Dear (Mr Y)) (Honour (Promote ProjectManager))
```

Now the linearization knows how to inflect the relevant parts in agreement with the new recipient, which results in the letter

> ***Cher** Monsieur Y, j'ai l'honneur de vous informer que vous avez été **promu** **chargé** de projet.*

Boldface is here used for marking the parts that have changed as a result of agreement and are now correct.

At the end of the previous section, we identified disambiguation as a critical part of interactive systems. Parsing user input may lead to several trees from which only the user is able to choose. The simplest way to display the alternatives is to show the abstract syntax trees, but this is hardly user-friendly, and we want the translation system to be usable without awareness of the abstract syntax. An alternative is to use a *disambiguation grammar*—a grammar that is similar to that of the source language, but contains supplementary information that makes it unambiguous.

Consider, for example, the disambiguation needed in the example with *the donkey ate the banana*. Since no humans are involved, the English grammar for pronouns could be simply

```
lin Pron _ _ = "it"
```

A disambiguation grammar for translation into German should make at least the type explicit.

```
lin Pron t _ = "it" ++ "(" ++ "the" ++ t ++ ")"
```

Hence, the question posed to the user when translating *it had fallen from the tree* would display the menu items: *it (the donkey)* and *it (the banana)*. Full disambiguation would of course also show the referent:

```
lin Pron t r = "it" ++ "(" ++ "the" ++ t ++ r ++ ")"
```

As illustrated by the MOLTO Phrasebook (Angelov & al. 2010), disambiguation grammars can be constructed with minor additions to the original, ambiguous grammars.

6 Variations in Concrete Syntax

How is it possible for different languages to share an abstract syntax? We have mainly considered two ways of achieving this: in different concrete syntaxes,

- *words* can be different;
- the *order* of words can be different.

However, more freedom is needed to enable an abstract syntax really to abstract away from language-dependent facts. Fortunately, two more things have proven to be enough to achieve this:

- *parameters*: words and phrases can have different inflectional forms and features;
- *discontinuity*: the translation of a word can consist of separate parts.

Let us first consider the parameters. English verbs (with the exception of *be*) have five forms, exemplified by *write, writes, wrote, written, writing*. German verbs have at least 20 finite forms, and moreover dozens of adjectival forms of the participles. French verbs have 51 forms, Latin verbs a couple of hundreds, Finnish verbs several thousands depending on how one counts. The grammar of each language has to define precisely how these forms are created for each verb and how they are used in sentences. Yet we want to have, in the abstract syntax, a common category of verbs, and common rules (i.e. functions) for combining verbs with their subjects and objects.

Here is an example: a function that forms a sentence (S) by combining a two-place verb (V2) with a subject and an object. The subject and the object are *noun phrases* (NP), such as pronouns (*she*), proper names (*Mary*), or nouns with determiners (*the banana*). The abstract syntax thus has three categories and one function:

```
cat S ; V2 ; NP
fun PredV2 : V2 -> NP -> NP -> S
```

To achieve a complete description in little space, let us restrict the grammar to present indicative sentences. The simplest possible concrete syntaxes are

```
lin PredV2 v s o = s ++ v ++ o
lin PredV2 v s o = s ++ o ++ v
```

and four other permutations; thus there is no problem to treat so-called SVO and SOV languages with the same abstract syntax. Swedish is almost as simple as this, since the verb has just one form for the present indicative. However, pronouns have separate nominative and accusative forms, used for the subject and the object, respectively. Thus for Swedish, we have to change the *linearization type* of noun phrases from strings to *tables*, which assign a string to each of the cases nominative and accusative. We write

```
lincat NP = Case => Str
```

to say that noun phrases are linearized to case-to-string tables. We write

```
param Case = Nom | Acc
```

to define the *parameter type* of cases in Swedish. And finally, we write

```
lin PredV2 v s o = s ! Nom ++ v ++ o ! Acc
```

to linearize subject-verb-object sentences in Swedish. The *selection* operator `!` is used for retrieving values from tables. The tables themselves are given with a special expression form, as shown in the rule for the pronoun *she*:

```
fun She : NP
lin She = table {Nom => "hon" ; Acc => "henne"}
```

German is more complex than Swedish in three ways: noun phrases have four cases instead of two; verbs have five forms instead of one; and the object can have different cases depending on the verb. For instance, *lieben* (“love”) takes its object in the accusative, but *folgen* (“follow”) in the dative. We need a more complex system of parameters and linearization types:

```
param Case = Nom | Acc | Dat | Gen
param Number = Sg | Pl
param Person = Per1 | Per2 | Per3
lincat NP = {s : Case => Str ; n : Number ; p : Person}
lincat V2 = {s : Number => Person => Str ; c : Case}
```

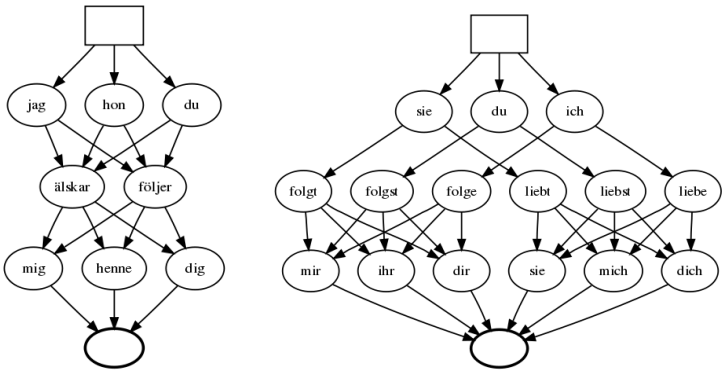
These linearization types use yet another data structure provided by GF: *records*. A record is a collection of objects of possibly different types. Thus the German NP has a case-to-string table, a number, and a person. These objects can be retrieved from the record by the *projection* operator `.` (dot). Now we can write a German linearization rule that selects the correct forms of the subject, the verb, and the object:

```
lin PredV2 v s o = s.s ! Nom ++ v.s ! s.n ! s.p ++ o.s ! v.c
```

Even though this rule is more complex than the Swedish rule, the abstract syntax is still the same. To see how much they differ, consider the language fragment generated from two verbs and three pronouns. The abstract syntax trees are given by the expression

```
PredV2 (Love|Follow) (I|You|She) (I|You|She)
```

where | marks alternatives. Thus there are 18 different trees (which include questionable ones such as *I love me*). The following picture shows finite automata representing the concrete syntaxes, Swedish on the left and German on the right. The German automaton is more complex. One can notice, for instance, that the number of different words (word forms) is almost twice the number of word forms in Swedish (14 vs. 8).



The higher the number of word forms in a language, the less probable is the occurrence of each word in a corpus. This is a problem for statistical string-based language models. For instance, a system may fail to find some less common German verb forms at all. A remedy to this is to introduce grammatical knowledge into the system by analysing the words into their dictionary forms and morphological description tags. Thus for instance

```
du folgst mir
```

becomes something like

```
du<+Pron+Nom> folgen<+Verb+Ind+Pres+Sg+2> ich<+Pron+Acc>
```

Then an *n*-gram model can be built for the sequences of description tags, which are much more frequent than the verb forms, and the translations of dictionary forms can be defined separately. This idea is known as *factored translation models* and studied in Koehn and Hoang (2007).

A prime example of *discontinuous constituents* are the compound verbs of Germanic languages. Thus the German verb *umbringen* (“kill”) consists of the verb *bringen* (“bring”) and the particle *um* (“around”). In the sentence

er bringt mich um

("he kills me", word to word "he-brings-me-around"), these parts fit into one and the same 3-gram, whereas in

er bringt meinen besten Freund um

("he kills my best friend") the chances are that a purely statistical system misses the whole point of the sentence.

In GF, discontinuous constituents can be modelled with records. Thus we extend the German linearization type of V2 with a field for the particle:

```
lincat V2 = {s : Number => Person => Str ; p : Str ; c : Case}
```

The predication rule becomes correspondingly

```
lin PredV2 v s o =
  s.s ! Nom ++ v.s ! s.n ! s.p ++ o.s ! v.c ++ v.p
```

The full grammar of subject-verb-object predication is of course much more complex than shown above. In German, we have to take into account the word order variation in main clauses and subordinate sentences. In French, the position of the object is different for pronouns from heavier noun phrases (*je t'aime* "I-you-love" vs. *j'aime Marie* "I-love-Marie"), with subtle agreement differences in compound tenses, and so on. Nevertheless, the two data structures of GF (tables and records) have proven sufficient for the expression of all these rules in a compact and efficient manner. Thus the *GF Resource Grammar Library* (Ranta 2009a) covers a large fragment of syntax of 16 languages using the same abstract syntax for all languages. As we will see in next section, this library plays a key role in the economical production of translation systems.

Interestingly, the GF Resource Grammar Library also demonstrates that the morphology and syntax, despite huge differences in surface variation, has a similar complexity in each language. This is not only reflected in the shared abstract syntax, but also in the size of the source code. The following table shows the code size for the concrete syntaxes of one and the same abstract syntax for four languages in the library, together with relative standard deviations and the maximum-to-minimum ratios. While the "GF source" column shows little variation, the low-level generated code rules varies much more. The "compressed PGF" column gives the size of the binary code generated by the GF compiler to implement parsing and linearization at run time, compressed by bzip2. The "context-free" column gives the number of rules in a context-free grammar generated as a conservative approximation of the PGF. The "words" column gives the total number or words in the PGF, optimized by a restriction to the words reachable from the start category.

language	**GF source**	**compressed PGF**	**contex-free**	**words**
English	3300	49000	48000	1900
German	3400	67000	63000	4400
French	5400	84000	66000	4100
Finnish	4200	103000	192000	21000
rel. stdev.	0.21	0.26	0.63	0.97
max/min	1.9	2.1	4.0	11

The numbers of context-free rules and words reflect the complexity of the language on a low abstraction level, whereas the GF code reflects the amount of information needed for defining the language on a high abstraction level. The PGF size also involves an abstraction in the sense of redundancy reduction due to compression and underlying compiler optimizations such as common subexpression elimination.

7 Grammar Engineering

The figures at the end of the previous section suggest that implementing a grammar in GF is not significantly harder for a "complex" language like Finnish than for a "simple" language like English. This result holds for grammars that exploit all the abstractions available in GF. If the grammars had to be written in context-free format, the differences would be much larger. If the language models had to be constructed from the occurrences of words, German and French would need more textual data than English, and Finnish would need even more. This is illustrated by the fact that a Google translate from Finnish often returns Finnish word forms untranslated.

Despite the compactness of GF, grammar writing is not easy. It requires a lot of linguistic knowledge, and only a part of this can be found in standard reference books, in particular as regards syntax. These difficulties may have been a major obstacle to the popularity of grammar-based domain-specific precision-oriented translation systems. While good quality can be reached with the use of grammars on limited domains, writing these grammars is time-consuming. Moreover, it requires the joint competences of a linguist and a *domain expert*, for instance, a mathematician when building a translation system for mathematical texts, or a car engineer when translating car maintenance manuals.

When the first multilingual GF grammars were built for a few domains (mathematics, tourist phrasebooks, restaurant database queries, medical drug descriptions), it soon become obvious that the same linguistic problems arose over and over again. This was the main reason for starting to build the GF Resource Grammar Library (cf. Ranta 2009b). Now that the library is complete for a large fragment of language, it is clearly the most important factor in enhancing the productivity in building translation systems. The library has a high-level API (*Application Programmer's Interface*), which corresponds to an abstract syntax and hides the details like word order, inflection, agreement, and discontinuities. The API is, moreover, language-independent, so that the grammar code for one language can also be used for other languages covered by the library.

To take an example of the use of the library, consider a concept such as *x knows y*, a two-place relation between persons. This might be needed in a translation system for social fora, and defined by the abstract syntax predicate

```
fun Know : Person -> Person -> Fact
```

The concrete syntax can be defined by means of the resource grammar by using NP (noun phrase) as the linearization type of persons and Cl (clause) as the linearization type of facts. The library API displays a function

```
mkCl : NP -> V2 -> NP -> Cl
```

for building a clause from a subject, a two-place verb, and an object. The library moreover provides language-specific lexica of irregular verbs. For instance, the English library has a constant

```
know_V : V
```

There is also a functions for making a verb (V) into a transitive two-place verb (V2),

```
mkV2 : V -> V2
```

Now we can define

```
lin Know x y = mkCl x (mkV2 know_V) y
```

This rule produces all the variation that can occur in a clause. Some of the variations are due to agreement to the subject:

```
Know I She  --> I know her
Know He She --> he knows her
```

but much more is needed when the clause is put into a wider context, such as negation, questions, and tenses:

```
mkS negative_Pol (Know I She)         --> I don't know her
mkQS (Know I She)                     --> do I know her
mkS past_Tense (Know I She)           --> I knew her
mkS fut_Tense anter_Ant (Know I She) --> I will have known her
```

In all these examples, we have just wrapped the clause `Know I She` with resource grammar functions to produce the correct linearizations.

The same API works for German and French, by just changing the verb:

```
lin Know x y = mkCl x (mkV2 kennen_V) y
lin Know x y = mkCl x (mkV2 connaître_V) y
```

In addition to the variations listed above, German displays word order variations:

```
mkS if_Subj (mkS (Know She I)) (mkS (Know I She)) -->
  wenn sie mich kennt, kenne ich sie
```

("if she knows me I know her"). In French, clitic variations are produced:

```
Know I Marie --> je connais Marie
Know Marie I --> Marie me connait
```

In none of these cases does the user of the library need to know about word order, clitics, inflection, or agreement—she just has to decide what the verb is, select the subject and the object, and perhaps the tense or some other context where the clause is to be used.

A futher effectivization of grammar writing is provided by the use of *functors*. A functor, or a *parametrized module*, is a program module that depends on some undefined constants and can be instantiated by defining these constants. The previous example suggests a functor definition of the predicate *know*:

```
lin Know x y = mkCl x know_V2 y
```

where `know_V2` is an undefined constant. Each of the three languages define it separately:

```
know_V2 = mkV2 know_V
know_V2 = mkV2 kennen_V
know_V2 = mkV2 connaître_V
```

Technically, also `mkCl` is an undefined constant. Its definition for each language is given in the GF Resource Grammar Library. In general, functors use two kinds of constants:

- syntactic constants defined in the library
- lexical constants defined by the programmer

This way of using functors has become standard in GF projects. Grammarians that use the library thus have to write three kinds of modules:

- abstract syntax, to define the semantics of a new domain
- concrete syntax functor, to implement the first language on a new domain
- domain lexicon, to implement a new language in an old domain

The first two tasks demand domain expertise and knowledge about GF and the Resource Grammar Library, but no detailed linguistic knowledge. The third task demands little more than a native speaker's knowledge of the target language and the terminology of the domain.

The Resource Grammar Library thus minimizes the knowledge requirements for building translation systems: the programmer gives the words, and the grammar governing the words comes from the library. This technique is similar to how human speakers of a foreign language learn new words. If I know the grammar of German, but I don't know how to express a concept such as *x intersects y* in geometry, I can ask someone how to translate this very example. Then I can use my knowledge of German grammar to generalize the translation to more complex cases such as *wenn x nicht y schnitte, würde y auch nicht x schneiden* ("if x didn't intersect y, y wouldn't intersect x either").

The translation of the sentence *x intersects y* as *x schneidet y* can actually be interpreted as the linearization rule

```
lin Intersect x y = mkCl x (mkV2 schneiden_V) y
```

if the sentence *x schneidet y* can be parsed in the German resource grammar as a tree of type Cl. This suggests a method for *example-based grammar writing*, where the library is not used explicitly but via examples, which in this case could be given in a format such as

```
lin Intersect x y = parse Cl "x schneidet y"
```

The crucial piece of information is here the German string. One way to obtain this string is by giving the English string to a human translator, who thus doesn't need any knowledge of GF; she may need to know that the context is that of geometry, to resolve potential word sense ambiguities.

By example-based grammar writing, building a translation system that can deal with an unlimited number of documents boils down to translating just one document, which contains representative examples of all concepts in the domain. This is probably one of the most efficient ways to use human labour in machine translation.

An extreme form of example-based grammar writing is to give the translation examples to a statistical machine translator system, such as Google translate. In fact, properly trained statistical translators are quite reliable with sentences that are short and typical (i.e. frequent n-grams for a small n). Thus, for instance, Google translate gets *x intersects y* right in German. Since the produced GF rule covers all variation due to agreement, tense, and word order, it expands to 288 context-free rules. Most of these derived combinations are not translated correctly in Google translate—but this doesn't matter, since we now have a grammar-based system. In analogy to what we concluded about human labour, providing translations for example-based grammar writing might be one of the most reliable ways of using statistical models in machine translation.

8 Transfer and Paraphrasing

We have presented GF as a framework for interlingua-based translation systems. The need of transfer functions (i.e. functions mapping source language trees to target language trees) is less common than in some other grammar formalisms, because the abstract syntax trees of GF can maintain a considerable distance to the concrete trees.

In traditional systems, transfer is used whenever there is a *structural change* between the source and the target language. A typical example is *my name is Bond*. In the English sentence, the subject is *my name*. In the German equivalent, *ich heiße Bond*, the subject is *ich* ("I"), and a special verb *heißen* ("have name") is used. The French translation is *je m'appelle Bond*, literally "I call myself Bond".

In GF, translation is performed via the abstract syntax of a semantic grammar, rather than the syntactic resource grammar. If the abstract syntax has a predicate

```
fun Named : Person -> Name -> Fact
```

it is not a problem to pick different syntactic structures as linearizations,

```
fun Named x y = mkCl (possessive x (mkN "name")) (mkNP y)
fun Named x y = mkCl x (mkV2 heißen_V) (mkNP y)
fun Named x y = mkCl x (mkV2 (reflV appeler_V)) (mkNP y)
```

(Notice that GF provides overloading for sets of functions that have different types, here `mkCl`.) Thus in the run-time translation, no transfer is needed—just parsing and linearization via the interlingua tree. The use of different syntactic structures (as defined by the resource grammar) in linearization rules has the effect of *compile-time transfer*.

It eliminates the need of run-time transfer and hence maintains the simple interlingua-based translation model of GF.

The only restriction that GF's interlingua model poses to translation is *compositionality*. More precisely, all linearization rules in GF are compositional, which means that the linearization of every tree must be defined as a function of the *linearizations* of its immediate subtrees (and not of the trees themselves). Using the notation of *t** for the linearization of a tree *t* and *f** for the linearization function of a function *f*, compositionality means that

$$(f\, t_1\, \ldots\, t_n)^* \;=\; f^*\, t_1^*\, \ldots\, t_n^*$$

If linearizations were just strings (as in context-free grammars), it would be impossible to maintain compositionality even in simple translation tasks. But the use of records and tables in GF makes it maintainable in most cases, and it requires some effort to find counterexamples.

One counterexample to compositinal translation is suggested by the *my name is* example above. In German, the predicate has the bearer of the name as its subject, and the subject can be shared by *verb phrase coordination*:

ich heiße Bond und komme aus England

("I have-name Bond and come from England"). The English translation is

my name is Bond and I come from England

But this translation has changed the original's conjunction of predicates (verb phrases) to a conjunction of sentences, because there is no common subject that could be shared. On the abstract syntax tree level, it involves a transfer from a tree of the form

```
PredVP a (ConjVP F G)
```

to a tree of the form

```
ConjS (PredVP a F) (PredVP x G)
```

Also run-time transfer functions can be defined in GF. They are executed on the abstract syntax level between the source text parser and the target text generator. In general, translation from L_1 to L_2 is then a composition of three operations,

$$\text{parse}\, L_1 \implies \text{transfer}\, L_1\, L_2 \implies \text{linearize}\, L_2$$

Interlingual translation is a limiting case, where transfer is the identity mapping. Notice that the role here assigned to transfer is very similar to the operations that *optimizing compilers* such as GCC perform on the intermediate tree language level.

Transfer involves a departure from the interlingual model, destroys reversibility, and may compromise run-time efficiency. Therefore translators written in GF have usually avoided it. However, it can be useful to look at some of the fundamental properties suggested by type theory.

Like many logical frameworks, the abstract syntax part of GF can define a notion of *definitional equality* among syntax trees. For instance, the correspondence between sentence and verb phrase conjunction can be seen as a definitional equality. A reasonable

condition for transfer functions is that they must preserve definitional equality. Definitional equality is generated by arbitrary recursive function definitions, which need not be compositional.

The linguistic counterpart of definitional equality is *paraphrasing*. Two concrete syntax expressions are paraphrases, if their syntax trees are definitionally equal. We could say that the *literal translation* of a string is the one obtained by parsing the string and linearizing the resulting tree *t*. If the tree is first converted to a definitionally equal tree *t'*, we obtain a *translation by paraphrase*.

In general, there can be infinitely many trees definitionally equal to a given tree. These trees can be generated by applying equality rules forwards and backwards. To take the most familiar example from type theory, assume the set of natural numbers defined by the constructors

```
data Zero : Nat
data Succ : Nat -> Nat
```

(the keyword `data` in GF marks a function as a constructor). Then add the constant 1 and the addition operation, with their definitions, written as follows in GF:

```
fun one  : Nat
def one  = Succ Zero

fun plus : Nat -> Nat -> Nat
def plus x Zero = x
def plus x (Succ y) = Succ (plus x y)
```

The tree `one` now has infinitely many paraphrases, beginning with those obtained in one computation step forward or backward,

```
Succ Zero, plus one Zero, plus (Succ Zero) Zero, ...
```

The *computation distance* of a tree *t'* from a tree *t* is the number of steps needed to obtain *t'* from *t*. A possible condition for translation by transfer is that it should minimize the computation distance. (An alternative measure would be *tree edit distance*, but computation distance has the advantage that the optimal sorting of trees can be straightforwardly generated from the definitions.)

Now, the minimal computation distance is 0, and this cannot always be achieved because there are other constraints. One such constraint is the concrete syntax of a target language can simply lack the construct used in the source language. Another constraint, reasonable in domains that require precision, is *non-ambiguity*: if a string is unambiguous in the source language, its translation should not be ambiguous in the target language. This condition is decidable relative to a fixed grammar, because the GF parser can find all trees corresponding to a string (except in some pathological cases where the number of trees is infinite).

The transfer task can therefore be defined as finding the closest unambiguous paraphrase. A typical example is anaphora. For instance, Finnish has a gender-neutral pronoun, *hän*, corresponding to both *he* and *she*. Therefore a love story written in English often has to paraphrase *he* with *mies* (“the man”) and *she* with *nainen* (“the woman”).

The optimal translation is more subtle than this: consider a context in which a man and a woman are given, and the sentence

She took his hand.

There are two equally close unambiguous translation of the second sentence,

Hän tarttui miehen käteen. ("She took the man's hand.")
Nainen tarttui hänen käteensä. ("The woman took his hand.")

One does not need to paraphrase both pronouns to remove ambiguity, because the other pronoun can only refer to a different person. If it was the same person, the object position would be rendered as a reflexive.

Yet another ingredient in transfer is *style*. Some constructs used in a source language can be unnatural in the target language, even if they would be possible and unambiguous. For instance, passive constructions with agents are common in English, but are preferably translated by active constructions in Finnish. One way to implement this in GF is to assign *weights* to abstract syntax functions, reflecting their goodness in each target language. The weight of a tree is then the product of the weights of the functions in all of its nodes. Transfer should then find the paraphrase that has the maximal weight in the target language, at the same time as minimizing the computation distance to the literal translation and maintaining non-ambiguity.

One way in which weights can be assigned to abstract syntax functions are via their relative frequencies in some corpus. This leads to the notion of *probabilistic GF grammars* and shows yet another way in which statistical language models can be combined with grammars.

9 Specification and Evaluation with Grammars

Translation from one language to another is a function ultimately defined on the level of strings: input a string in the source language, output a string in the target language. But the type `String -> String` is of course too lean as a specification of what translation should do, and must be refined. The standard refinement in the statistical translation community is the *BLEU score* (Papineni 2002), which compares the output of a translating system with some *gold standard* translation produced by a human. The BLEU score is computed by counting the occurrences of words and n-grams, taking into account their order. The best translation is one that matches the gold standard word by word. But also some intuitively bad translations, such as one that matches the original word by word but forgets a negation word, get high scores. And intuitively excellent translations by humans get bad scores, if they don't use the same words as the gold standard.

The problems with the BLEU score are widely acknowledged, but it is still popular because it is automatic. Since the goal of statistical translation systems is often defined as the maximization of the BLEU score, one of its main uses is in the development phase of the systems. A typical system uses several *features* to compute the most likely translation: frequencies from bilingual word alignment, frequencies of n-grams in the source language, etc. Each of these features is given some weight, and the system is tested with

several distributions of weights to maximize the BLEU score. While this makes sense, using the BLEU score as a measure of the quality when comparing translation systems is less adequate.

So what alternatives are there for specifying what a translation function should do? The traditional answer is, of course, that the translation should preserve meaning. An implicit presupposition is that it should render grammatically correct output; otherwise it would not count as a target-language expression at all! Fluency and good style (or style that matches the style of the source) are further requirements. For most of these criteria, there is no other evaluation method than human judgement. However, the GF-based method described in this paper suggests a technique for cross-evaluating other systems (such as statistical ones). Assuming that grammaticality is properly defined in GF, GF grammars can be used for assessing the grammaticality of the output from these other systems. The same concerns the preservation of meaning, if we have a GF grammar in which the linearizations and definitional equalities preserve meaning.

A problem in using GF to evaluate statistical translation systems is of course that the coverage of GF is just partial. But it can still be used to test the quality of a statistical translator for those sentences that the GF grammar does cover. The relevant functionality of GF is *multilingual generation*: one can produce a *synthesized corpus*, that is, a set of sentence pairs in a source and a target language, where the sentences in each pair have the same abstract syntax tree. This set can then be used for evaluating a statistical translator, because it gives both the source and the gold standard translations.

Multilingual generation also provides a way to build a statistical translator in the first place. A GF grammar can be used for producing any number of aligned sentences, which can be guaranteed to cover all word forms appearing in the grammar. All sentence pairs can be automatically equipped by correct phrase alignments, which is a good starting point for building the statistical model (Och and Ney 2004). Such an alignment relates to each other the words that have the same smallest spanning subtree in the abstract syntax. The alignments can cross, and they can include many-to-many relations, as shown in the figure in Section 1, generated from the GF Resource Grammar Library.

The resulting statistical model may perform reasonably for the input covered by the grammar, although never better than the grammar itself. However, the advantage of the statistical model is that it is also able to translate sentences not recognized by the grammar. In this way, the statistical model can be used as a smoothing technique for grammar-based translation. Of course, it is then important to mark clearly which parts of the output are translated by smoothing and which parts come from the grammar. This technique is analogous to the use of GF-generated language models for speech recognition studied by Jonson (2006). The result is a hybrid system where a statistical model is derived from a grammar, and the grammar can in turn have been built as a generalization of a less refined statistical model as described in Section 7. An experiment with this idea can be found in Bouillon and Rayner (2011).

10 Conclusion

Machine translation was attempted as one of the first applications of digital computers. It was soon realized that fully automatic high-quality translation is impossible. The

main conclusion drawn from this was that there is a trade-off between *coverage* and *precision*. The efforts on machine translation are thus roughly divided into open-domain systems aiming at coverage and closed-domain systems aiming at precision. In both kinds of systems, human interaction can be involved. In open-domain systems, a typical form of interaction is the post-processing of the output to improve its quality. In closed-domain systems, a typical form of interaction is to ask a human to disambiguate. One can also use human interaction to recover from input that is not in the grammar, for instance, to add the translations of unknown words.

The use of type theory in machine translation dates back to the earliest years, when Bar-Hillel applied it to the formal representation of grammar and meaning. GF is a contemporary variant of the idea, providing a notion of multilingual grammars, a framework for applying the method to new domains, a resource grammar library for improving the productivity for new languages, and a set of user interface components (parsing, syntax editing) to help the work of translators.

GF has since 1998 been used for translation on several domains, including mathematics (Hallgren and Ranta 2000, Caprotti 2006), software specifications (Johannisson 2005), and spoken dialogue systems (Bringert & al. 2005, Perera and Ranta 2007). The European MOLTO project (Multilingual On-Line Translation) aims to scale up the methods into larger domains, more languages, easier production and application, and more robustness (i.e. recovery from out-of-the grammar input).

Some of the GF methods presented in this paper have not yet been used in actual translation systems. On the purely type-theoretical side, these include the anaphora resolution algorithm (Section 4) and the generation of paraphrases via definitional equality (Section 8). On the hybrid side, the ranking of paraphrases (Section 8) and the training of statistical translations (Section 9) have just had their first experiments in the MOLTO project. On the other hand, example-based grammar writing (Section 7) and disambiguation grammars (Section 5) are recent ideas both of which have given promising results in the first demonstrator of MOLTO, which is a phrasebook for translating touristic phrases between 14 languages (Angelov & al. 2010).

The first experiences with the Phrasebook and with a translator of mathematical exercises are confirming the basic tenet of MOLTO: that it is possible to build reliable translation systems for limited domains by careful engineering and adequate tools. With almost any language pair and example covered by the grammar, the translation quality is better than the quality produced by general-purpose statistical systems. In fact, it can be better in a crucial way, since grammar-based translation can be guaranteed to be correct by design whereas statistical systems always involve an element of uncertainty.

On the other hand, if we want a system that automatically translates any input, uncertainty cannot be avoided, and statistical methods are the ones that *de facto* yield the best quality in most cases. An interesting exception is translation between closely related languages, such as Swedish and Danish (Tyers and Nordfalk 2009). The lack of bilingual training data can make it impossible to build good statistical systems; at the same time, simple rules (such as replacing words by their equivalents in proper forms) may be sufficient to produce very good quality.

Acknowledgements

Per Martin-Löf supervised my PhD thesis and taught me how to think about type theory and language. He also introduced me to the noisy channel model of Shannon. He wondered if statistical models were still considered useful in natural language processing, and they have ever since been a recurrent theme in our discussions. With his unique combination of insights in both statistics and logic, and his accurate knowledge of many languages, Per has continued to be a major resource for my work through the 21 years that have passed since my PhD. When I later started to look closer at statistical methods, I received inspiration and guidance from Joakim Nivre, Lluís Màrquez, and Cristina España. Lauri Carlson has helped me to understand the problems of translation in general. The model described in this paper has received substantial contributions from my own PhD students Peter Ljunglöf, Kristofer Johannisson, Janna Khegai, Markus Forsberg, Björn Bringert, Krasimir Angelov, and Ramona Enache. The insightful comments from an anonymous referee were valuable when preparing the final version of the paper. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n:o FP7-ICT-247914 (http://www.molto-project.eu).

Bibliography

Ajdukiewicz, K. (1935). Die syntaktische konnexität. *Studia Philosophica 1*, 1–27.

Alshawi, H. (1992). *The Core Language Engine*. Cambridge, Ma: MIT Press.

Angelov, K. (2009). Incremental Parsing with Parallel Multiple Context-Free Grammars. In *Proceedings of EACL'09, Athens*.

Angelov, K., O. Caprotti, R. Enache, T. Hallgren, I. Listenmaa, A. Ranta, J. Saludes, and A. Slaski (2010, 06/2010). D10.2 molto web service, first version. (D10.2).

Angelov, K. and R. Enache (2010). Typeful Ontologies with Direct Multilingual Verbalization. In N. Fuchs and M. Rosner (Eds.), *CNL 2010, Controlled Natural Language*.

Appel, A. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press.

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language 29*, 27–58.

Bar-Hillel, Y. (1964). *Language and Information*. Reading, MA: Addison-Wesley.

Bringert, B., R. Cooper, P. Ljunglöf, and A. Ranta (2005, June). Multimodal dialogue system grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*, pp. 53–60.

Brown, P. F., J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin (1990). A statistical approach to machine translation. *Computational Linguistics 16*(2), 76–85.

Burke, D. A. and K. Johannisson (2005). Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In P. Blache and E. Stabler and J. Busquets and R. Moot (Ed.), *Logical Aspects of Computational Linguistics (LACL 2005)*, Volume 3492 of *LNCS/LNAI*, pp. 51–66. Springer. http://www.springerlink.com/content/?k=LNCS+3492.

Caprotti, O. (2006). WebALT! Deliver Mathematics Everywhere. In *Proceedings of SITE 2006. Orlando March 20-24*. http://webalt.math.helsinki.fi/content/e16/e301/e512/PosterDemoWebALT_eng.pdf.

Chandioux, J. (1976). MÉTÉO: un système opérationnel pour la traduction automatique des bulletins météreologiques destinés au grand public. *META 21*, 127–133.

Curry, H. B. (1961). Some logical aspects of grammatical structure. In R. Jakobson (Ed.), *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pp. 56–68. American Mathematical Society.

Donzeau-Gouge, V., G. Huet, G. Kahn, B. Lang, and J. J. Levy (1975). A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*.

Dowek, G., A. Felty, H. Herbelin, G. Huet, C. Parent, C. Paulin Mohring, B. Werner, and C. Murthy (1993). The Coq proof assistant user's guide : version 5.8. Research Report RT-0154, INRIA.

Dymetman, M., V. Lux, and A. Ranta (2000). XML and multilingual document authoring: Convergent trends. In *Proc. Computational Linguistics COLING, Saarbrücken, Germany*, pp. 243–249. International Committee on Computational Linguistics.

Hallgren, T. and A. Ranta (2000). An Extensible Proof Text Editor. In M. Parigot and A. Voronkov (Eds.), *LPAR-2000*, Volume 1955 of *LNCS/LNAI*, pp. 70–84. Springer. http://www.cse.chalmers.se/~aarne/articles/lpar2000.pdf.

Harper, R., F. Honsell, and G. Plotkin (1993). A Framework for Defining Logics. *JACM 40*(1), 143–184.

Hutchins, W. J. and H. L. Somers (1992). *An Introduction to Machine Translation*. London: Academic Press Limited.

Jelinek, F. (2009). The dawn of statistical ASR and MT. *Computational Linguistics 35*(4), 483–494.

Johannisson, K. (2005). *Formal and Informal Software Specifications*. Ph. D. thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University.

Jonson, R. (2006). Generating statistical language models from interpretation grammars in dialogue system. In *Proceedings of EACL06, Trento, Italy*.

Kay, M. (1997). The Proper Place of Men and Machines in Language Translation. *Machine Translation 12*(1–2), 3–23.

Khegai, J., B. Nordström, and A. Ranta (2003). Multilingual Syntax Editing in GF. In A. Gelbukh (Ed.), *Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico City, February 2003*, Volume 2588 of *LNCS*, pp. 453–464. Springer-Verlag. http://www.cs.chalmers.se/~aarne/articles/mexico.ps.gz.

Knuth, D. (1968). Semantics of context-free languages. *Mathematical Systems Theory 2*, 127–145.

Koehn, P. and H. Hoang (2007). Factored translation models. In *EMNLP-CoNLL*, pp. 868–876. ACL.

Ljunglöf, P. (2004). *The Expressivity and Complexity of Grammatical Framework*. Ph. D. thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University. http://www.cs.chalmers.se/~peb/pubs/p04-PhD-thesis.pdf.

Ljunglöf, P., G. Amores, R. Cooper, D. Hjelm, O. Lemon, P. Manchón, G. Pérez, and A. Ranta (2006). Multimodal Grammar Library. TALK. Talk and Look: Tools for Ambient Linguistic Knowledge. IST-507802. Deliverable 1.2b. http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/TK_D1-2-2.pdf.

Lopez, A. (2008). Statistical machine translation. *ACM Comput. Surv. 40*(3).

Luo, Z. and P. Callaghan (1999). Mathematical vernacular and conceptual well-formedness in mathematical language. In A. Lecomte, F. Lamarche, and G. Perrier (Eds.), *Logical Aspects of Computational Linguistics (LACL)*, Volume 1582 of *LNCS/LNAI*, pp. 231–250.

Luo, Z. and R. Pollack (1992). LEGO Proof Development System. Technical report, University of Edinburgh.

Magnusson, L. (1994). *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph. D. thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Napoli: Bibliopolis.

Montague, R. (1974). *Formal Philosophy*. New Haven: Yale University Press. Collected papers edited by Richmond Thomason.

Nordström, B., K. Petersson, and J. Smith (1990). *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford: Clarendon Press.

Norell, U. (2007, September). *Towards a practical programming language based on dependent type theory*. Ph. D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Och, F. J. and H. Ney (2004). The alignment template approach to statistical machine translation. *Computational Linguistics 30*(4), 417–449.

Papineni, K., S. Roukos, T. Ward, and W.-J. Zhu (2002). BLEU: a method for automatic evaluation of machine translation. In *ACL*, pp. 311–318.

Perera, N. and A. Ranta (2007). Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*. http://www.cs.chalmers.se/~aarne/articles/perera-ranta.pdf.

Pierce, J. R., J. B. Carroll, and al. (1966). Language and Machines — Computers in Translation and Linguistics. ALPAC report.

Power, R. and D. Scott (1998). Multilingual authoring using feedback texts. In *COLING-ACL 98*, Montreal, Canada.

Ranta, A. (1994). *Type Theoretical Grammar*. Oxford University Press.

Ranta, A. (2004). Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming 14(2)*, 145–189. http://www.cse.chalmers.se/~aarne/articles/gf-jfp.pdf.

Ranta, A. (2007). Modular Grammar Engineering in GF. *Research on Language and Computation 5*, 133–158. http://www.cs.chalmers.se/~aarne/articles/multieng3.pdf.

Ranta, A. (2009a). Grammars as Software Libraries. In Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin (Eds.), *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*, pp. 281–308. Cambridge University Press. http://www.cse.chalmers.se/~aarne/articles/libraries-kahn.pdf.

Ranta, A. (2009b). The GF Resource Grammar Library. *Linguistics in Language Technology 2*. http://elanguage.net/journals/index.php/lilt/article/viewFile/214/158.

Ranta, A. (2011). *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Rayner, M., P. Estrella, and P. Bouillon (2011). Bootstrapping a statistical speech translator from a rule-based one. In *Proceedings of the Second International Workshop on Free/Open-Source Rule-Based Machine Translation (2011: Barcelona)*. http://hdl.handle.net/10609/5647.

Rosetta, M. T. (1994). *Compositional Translation*. Dordrecht: Kluwer.

Shannon, C. (1948). A Mathematical Theory of Communication. *The Bell System Technical Journal 1*.

Stallman, R. (2001). *Using and Porting the GNU Compiler Collection*. Free Software Foundation.

Teitelbaum, T. and T. Reps (1981). The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM 24*(9), 563–573.

Tyers, F. and J. Nordfalk (2009). Shallow-transfer rule-based machine translation for Swedish to Danish. In *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation (2009: Alicante)*. http://hdl.handle.net/10045/12024.

Welsh, J., B. Broom, and D. Kiong (1991). A design rationale for a language-based editor. *Software: Practice and Experience 21*, 923—-948.