Computation as Subtyping: On the Turing Completeness of Type Systems, with Applications to Formal Grammars

Anonymous

ABSTRACT

Typed feature structures have a range of applications in logic and linguistics. Various formalisms exist, some including operations that are known to be Turing-complete, and others known to be less expressive. In this paper, we consider a restricted formalism which is used in a number of HPSG grammars, where the unification operation was believed to be of limited computational power. We show to the contrary that, even without mechanisms such as disjunctive constraints, the requirement that every substructure is well-typed is enough to yield Turing completeness. This relies on the fact that enforcing a type constraint can recursively enforce another type constraint on a substructure. To make such recursive type constraints easier to work with in practice, we introduce the concepts of "computation types" and "wrapper types", and give some illustrative examples, including specific operations on lists and booleans, as well as applications to several syntactic phenomena, including coordination, long-distance dependencies, valence changes, and free word order.

Keywords: type logic, feature structures, unification, computability, recursion, relational constraints

INTRODUCTION 1

Typed feature structures provide an effective way to organise information, and they are used in precise linguistic theories, including Head-driven Phrase Structure Grammar (HPSG; Pollard and Sag 1994). In

this paper, we follow the formalisation of typed feature structures presented by Copestake (2000), which has become the Joint Reference Formalism for the DELPH-IN consortium (for example, see: Bender *et al.* 2010; Fokkens *et al.* 2011; Bender and Emerson 2021; Zamaraeva *et al.* 2022). This formalism is supported by a number of processing engines, including the LKB (Copestake 2002), PET (Callmeier 2000), ACE (Crysmann and Packard 2012), and AGREE (Slayden 2012).

This formalism is deliberately restricted compared to some other versions of HPSG. All constraints are directly expressed as feature structures, without a separate notion of "feature structure description" (for an account of the historical development, see: Flickinger *et al.* 2021). For example, this rules out disjunctive constraints and relational constraints, which are available in the TRALE system (Meurers *et al.* 2002; Penn 2004). The set of types is finite, with no mechanism for defining "complex" types such as dot types (Pustejovsky 1995) or dependent types (Martin-Löf 1984). The only operations for defining a new feature structure are unification and generalisation.

The simplicity of the formalism might suggest that it has less computational complexity, and indeed, Copestake claimed that "the type inference system is essentially non-recursive". However, we show in this paper that the type inference system is in fact Turing-complete.

In §2, we review the formalism, and show that type inference can be recursive. To set up a practical framework for working with recursive type constraints, in §3 we explain how functions can be encoded as subtypes, giving examples on both finite domains (such as booleans) and infinite domains (such as natural numbers and lists). In §4, we quantify the computational complexity of the formalism, showing that one-feature type systems are finite-state, but two-feature type systems are Turing-complete. In §5, we present a general design pattern suitable for large-scale grammars, allowing a separation between the encoding of functions and their application in phrase structure rules, and we apply this design pattern to several syntactic phenomena.

2 FORMAL FRAMEWORK

In this section, we summarise the formalism presented by Copestake (2000), which we adopt in this paper. In brief, a type system consists

of a type hierarchy, a set of features, and a set of constraints. The type hierarchy and the features allow us to define feature structures, and the constraints control which feature structures are well-formed.

A type hierarchy is a finite upper-semilattice: a set equipped with a subsumption relation (a bounded-complete partial order), a unification operation (giving the greatest lower bound, if a lower bound exists) and a generalisation operation (giving the least upper bound).[1]

There are several equivalent ways of defining feature structures (for expositions, see: Carpenter 1992; Francez and Wintner 2011). Given a type hierarchy and a set of features, a typed feature structure can be defined as a rooted directed acyclic graph,[2] where nodes are labelled with types, edges are labelled with features, and edges from the same node must have distinct labels. Nodes can therefore be identified in terms of paths of features from the root. Hence, a typed feature structure F can be represented in terms of a partial function θ_F from paths to types, and an equivalence relation $\approx_F$ on paths indicating a re-entrancy (the paths lead to the same node).[3]

Subsumption on types naturally induces subsumption on feature structures: F subsumes G if it is more general in terms of both types and re-entrancies. More precisely, $F \sqsupseteq G$ if, for every path π in F, $\theta_F(\pi) \sqsupseteq \theta_G(\pi)$, and for every pair of paths, $\pi_1 \approx_F \pi_2$ implies $\pi_1 \approx_G \pi_2$.

To define a type system, each type t is assigned a constraint C_t, expressed as a feature structure with root of type t. More general types have more general constraints: if $t \sqsupseteq u$, then $C_t \sqsupseteq C_u$. A well-formed feature structure is one where all type constraints are satisfied: for each node, the substructure rooted at that node must be subsumed by the node's type's constraint. A type system is well-defined if all type constraints are well-formed (in other words, the constraints are compatible).

---

[1] Some authors follow the opposite convention, where: a hierarchy is a lower-semilattice, unification gives upper bounds, generalisation gives lower bounds.

[2] Acyclicity is not crucial for the results in this paper, although it sometimes serves to simplify a problem.

[3] Conversely, for a pair $(\theta, \approx)$ to be a feature structure, they must be compatible (if $\pi_1 \approx \pi_2$ then $\theta(\pi_1) = \theta(\pi_2)$), prefix-closed (if $\theta(\pi\alpha)$ is defined, so must $\theta(\pi)$), and fusion-closed (if $\pi_1 \approx \pi_2$ and $\theta(\pi\alpha)$ is defined, then $\pi_1\alpha \approx \pi_2\alpha$). If they are finite, then the feature structure is acyclic.

In addition, we can also introduce appropriateness constraints for features, assigning each feature α an appropriate type t_α. A well-formed feature structure must then also satisfy feature appropriateness: for each edge from each node, the node's type must be subsumed by the edge's feature's appropriate type. Feature appropriateness is not important for our complexity results in §3 and §4, but it will be essential for defining wrapper types in §5.

Finally, there are natural unification and generalisation operations on well-formed feature structures: unification gives the greatest lower bound (GLB), if a lower bound exists, and generalisation gives the least upper bound (LUB). In both cases, these are guaranteed to be unique.[4] If the unification of two feature structures is defined, they are said to be unifiable.

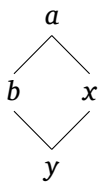
2.1 *Relational Constraints*

The formalism introduced above was intended to be computationally efficient. In particular, it does not include relational constraints, which would stipulate that the substructure rooted at one node must be related in some way to the substructure rooted at another node.

Aït-Kaci (1984) showed that relational constraints can be implemented using so-called "junk slots", which hold an intermediate step in a computation. However, implementing relational constraints in this way requires disjunctive type constraints. As an extension of the formalism presented above, this would mean that a type can be associated with more than one constraint structure, so that a well-formed feature structure is one where at least one constraint structure applies at each node. This is a flexible mechanism that enables relational constraints, as will be explained in more detail in §5.4.3, and contrasted with our proposed approach.

The lack of disjunctive constraints (and also other extensions) is what led Copestake (2000) to claim that the type inference system is non-recursive. We will see that relational constraints can in fact be implemented without disjunction, but using a different approach, as

[4] In contrast, this is not guaranteed in some other formalisms, for example if set-valued features are allowed (Pollard and Moshier 1990), or if constraints can be stated more expressively than as a feature structure (Carpenter 1992, §15).



$$a$$
$$b \xrightarrow{\mathrm{F}} a$$
$$x$$
$$y \xrightarrow{\mathrm{F}} b \xrightarrow{\mathrm{F}} x$$

Figure 1: A simple but pathological type system (left: hierarchy; right: constraints)

presented in §3. Before introducing this approach, it is instructive to first see a simple example of a type system exhibiting recursion, as demonstrated in the following section (§2.2).

Recursion: Pathological Counterexample 2.2

At first sight, it may seem that unification in our restricted formalism can be implemented straightforwardly: given two feature structures to be unified, we unify the re-entrancies and unify the types, then for any node with nontrivial type unification, we enforce the new type constraint. This is a deterministic process.

However, enforcing type constraints has the potential for recursion, which we will exploit in §3 and §4. In this section, we will illustrate the potential for recursion with a simple but pathological counterexample, a small type system where unification can fail to terminate, despite all type constraints being finite.

Consider the type system defined by the hierarchy and constraints in Fig. 1. The types a and b together allow the feature F to repeat any number of times. Pathological behaviour is created by the additional types x and y and the type constraint for y, as we will now see.

Consider unifying $y \xrightarrow{\mathrm{F}} b \xrightarrow{\mathrm{F}} x$ and $b \xrightarrow{\mathrm{F}} x$, both of which are well-formed. Combining the types and re-entrancies gives $y \xrightarrow{\mathrm{F}} y \xrightarrow{\mathrm{F}} x$, where the second node is of type y (unifying b and x). Enforcing y's constraint on that node then gives $y \xrightarrow{\mathrm{F}} y \xrightarrow{\mathrm{F}} y \xrightarrow{\mathrm{F}} x$, where the third node is of type y (again unifying b and x). This continues recursively.

Intuitively, the x or y types must repeat every two steps, exactly reaching the last node. The two input feature structures require this for even and odd numbers of steps, respectively, leapfrogging each other forever.

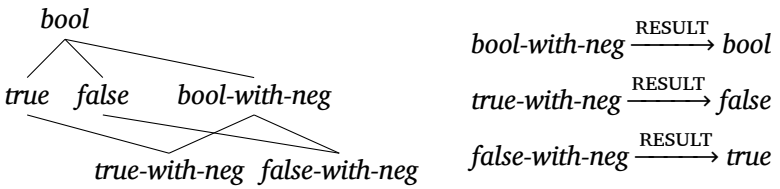


Figure 2: Type system for negation. For example, unifying *bool-with-neg* and *true* yields RESULT *false*.

3 COMPUTATION AS SUBTYPING

A function maps elements in some domain to elements in some range. However, a type system doesn't include functions as basic objects. In this section, we explain how to represent a function as a type, using a feature to hold the output of the function. Given a type system which includes the domain and range, we can define a larger type system which includes the desired function. This requires no change to the formalism: applying the function becomes a special case of unification, where the function type is unified with a type in the domain.

We will first discuss functions over finite domains in §3.1 (for example, boolean functions), and then turn to functions over infinite domains in §3.2 (for example, functions on natural numbers or lists).

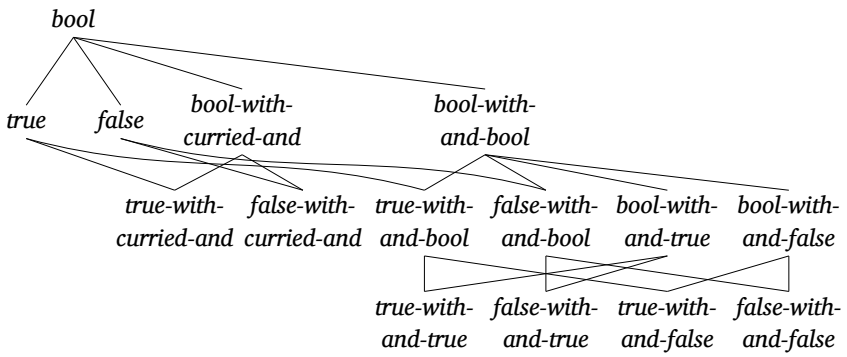
3.1 *Functions with a Finite Domain*

Consider a function from one type hierarchy to another.[5] The aim is to represent this function by defining a larger type hierarchy.

For each type in the domain, we define a new subtype, whose constraint structure includes the feature RESULT, whose value is the output of the function. Subsumption relations between the new subtypes are defined to exactly mirror those of the original hierarchy. An example is shown in Fig. 2, for logical negation (trivial constraints for *bool*, *true*, and *false* are not shown).

Formally speaking, there is nothing special about a type system defined in this way. However, the new subtypes and their constraints

[5] The function must respect the hierarchies: if $a \sqsupseteq b$ then $f(a) \sqsupseteq f(b)$. This is not restrictive: any function between finite sets can be extended to such a function, by adding an underspecified type to the domain and to the range.



$$\textit{bool-with-curried-and} \xrightarrow{\text{RESULT}} \textit{bool-with-and-bool} \xrightarrow{\text{RESULT}} \textit{bool}$$

$$\textit{true-with-curried-and} \xrightarrow{\text{RESULT}} \textit{bool-with-and-true} \xrightarrow{\text{RESULT}} \textit{bool}$$

$$\textit{false-with-curried-and} \xrightarrow{\text{RESULT}} \textit{bool-with-and-false} \xrightarrow{\text{RESULT}} \textit{false}$$

$$\textit{bool-with-and-bool} \xrightarrow{\text{RESULT}} \textit{bool}$$

$$\textit{bool-with-and-false} \xrightarrow{\text{RESULT}} \textit{false}$$

$$\textit{false-with-and-bool} \xrightarrow{\text{RESULT}} \textit{false}$$

$$\textit{true-with-and-true} \xrightarrow{\text{RESULT}} \textit{true}$$

Figure 3: Type system for logical "and", using currying (non-trivial type constraints shown below type hierarchy). For example, unifying *bool-with-curried-and*, *true*, and [RESULT *false*] yields [RESULT| RESULT *false*].

exactly represent the function f. Unifying the most general of the new subtypes with any original domain type x yields a structure where the value of RESULT is exactly $f(x)$.

The above procedure allows us to define unary functions. However, n-ary functions can be reduced to unary functions via "currying".[6] For example, a binary function $f : X, Y \to Z$ can be seen as a unary function $f : X \to (Y \to Z)$ mapping the first argument to a unary function of the second argument. The second function can be represented as a type, which will be the output of the first function. An example is shown in Fig. 3, for logical "and",[7] where type con-

[6] Named after Haskell Curry, but Curry credits the idea to Moses Schönfinkel.

[7] To satisfy feature appropriateness, we would need to add a type introducing the RESULT feature, which both *bool-with-curried-and* and *bool-with-and-bool* inherit from. This will be discussed further in §5.2.1.

Figure 4: Natural numbers from 0 to 3, represented as Peano feature structures.

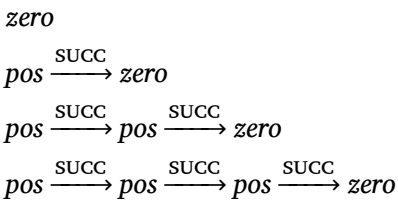


Figure 5: Type system for Peano feature structures.

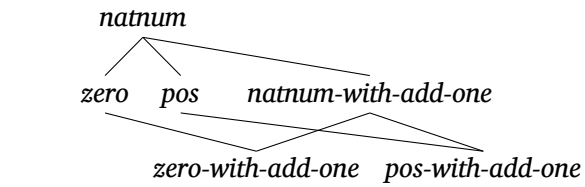
straints are only shown if they cannot be inferred from supertype constraints.[8] Defining all of the necessary subtypes of the second function effectively means defining a unary function for the Cartesian product of the input domains: for example, in the bottom row of the hierarchy in Fig. 3, there is one type for each combination of true and false in the two inputs. We will see a refinement of this idea in §5.3.1.

3.2 *Functions with an Infinite Domain*

An infinite domain cannot be directly represented as a type hierarchy, since the number of types must be finite. However, an infinite domain can be represented using typed feature structures, if there is recursion in the features. If a type admits an infinite set of well-formed feature structures (whose roots are subsumed by that type), then we can consider that type as representing an infinite domain.

Functions on an infinite domain can be defined in the same way as in §3.1, with the only difference that the constraints for the new subtypes can exploit the recursive features. The boolean hierarchy was finite and did not require any type constraints, but in the following sections we will consider natural numbers and lists, both of which can be defined as feature structures with recursive features.

[8] The type constraints can be defined even more succinctly: the values of RESULT|RESULT are redundant, since they can be inferred from other constraints. Fig. 3 shows well-formed constraints, rather than partial constraints that would be used in a practical grammar.



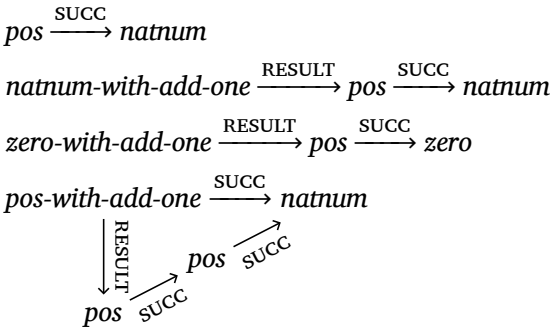


Figure 6: Peano type system, extended to include a function for adding one.

Natural Numbers 3.2.1

The simplest way to represent the natural numbers is using Peano numerals, where numbers above zero are recursively defined as "successors" of smaller numbers.

To implement Peano numerals in a type system, we can encode zero as an atomic type, and use a SUCC(essor) feature to encode larger numbers, with the number of SUCC features corresponding to the size of the number. This is illustrated in Fig. 4, where the *pos* type indicates a positive number. This means we can represent any natural number using just three types and one feature, as defined in Fig. 5.

For large numbers, this is an inefficient system (compared to binary or decimal, for example). However, its simplicity makes it useful for illustrating computation on an infinite domain.

In Fig. 6, the type system is extended to include a function that adds one to a number. As before, we define a subtype for each original type, and the type constraints together define the function's behaviour. For *pos-with-add-one*, the constraint uses a re-entrancy, so that the result has one extra SUCC feature, no matter the size of the input.[9]

[9] If cycles are allowed, we could make RESULT|SUCC re-entrant with the root. This would however mean the result is not "clean", a concept discussed in §5.2.1.

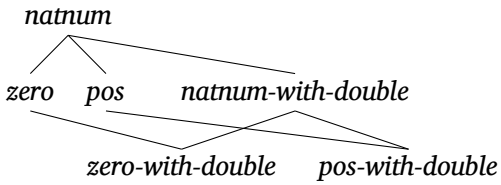
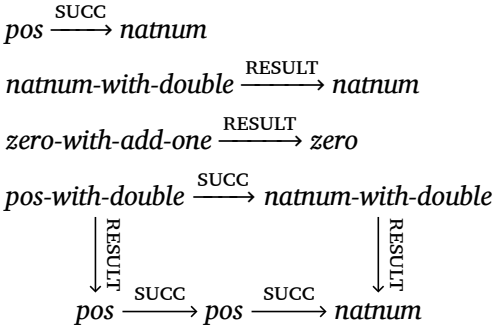


Figure 7: Peano type system, extended to include a function for doubling.



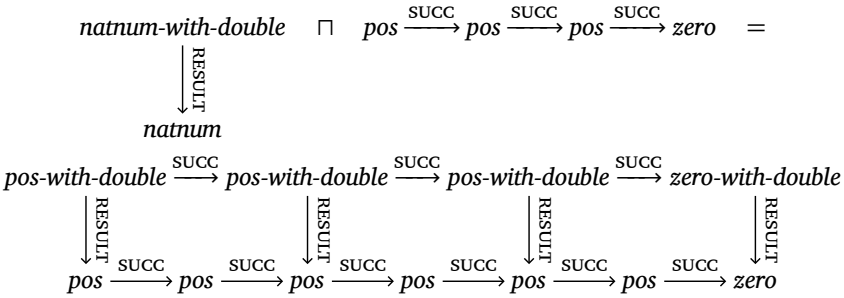


Figure 8: Example of unification with recursive type constraints, using the type system in Fig. 7.

As an example of a recursively defined function, we can consider doubling a number, as shown in Fig. 7. There are two crucial aspects to note in the constraint for *pos-with-double*. Firstly, the type *natnum-with-double* on SUCC means that the subtypes will be propagated along the whole structure. The more general type is used (*natnum-with-double* instead of *pos-with-double*) so that this propagation can terminate. Secondly, the re-entrancy between SUCC|RESULT and RESULT|SUCC|SUCC means that the value of the output is recursively defined based on the propagated subtypes. Since two SUCC features are added to RESULT for just one in the input, and since this process propagates along the whole input, the value of RESULT will

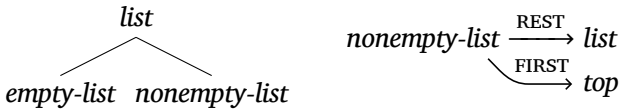


Figure 9: Types for lists.

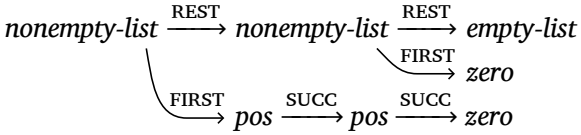


Figure 10: The list $\langle 2, 0 \rangle$ as a feature structure.

have exactly twice as many SUCC features as the input.

An example is shown in Fig. 8, where 3 is doubled to give 6. Similarly to the pathological type system that we saw in §2.2, this example illustrates how unification of well-formed structures can recursively trigger type constraints, leading to a much larger feature structure. Unlike the pathological type system, in this case the recursion eventually halts, when the propagated *natnum-with-double* types reach *zero*.

We will see this general pattern several times in this paper: propagation of subtyping along a feature, and re-entrancy between $\alpha|\text{RESULT}$ and $\text{RESULT}|\beta$, where α is a feature path in the input, and β is a feature path in the output.

Lists 3.2.2

Lists can be recursively defined in a type system, as shown in Fig. 9, using three types that mirror the ones we saw for Peano numerals. There are two features: REST is recursive, mirroring the SUCC feature, while FIRST holds an element of the list. The types in Fig. 9 are intended to be part of a larger type system, with *top* indicating the most general type in the larger hierarchy.

An example of a fully specified list is given in Fig. 10, where the list is of length two, containing the Peano numerals for two and zero.

As a simple example of a function on lists, the type system in Fig. 11 includes a function for calculating the length of a list. The *list-with-length* subtype propagates along the REST feature, and the re-entrancy between $\text{RESULT}|\text{SUCC}$ and $\text{REST}|\text{RESULT}$ recursively defines the output in terms of the input.

Lists will be further discussed in §5.4.

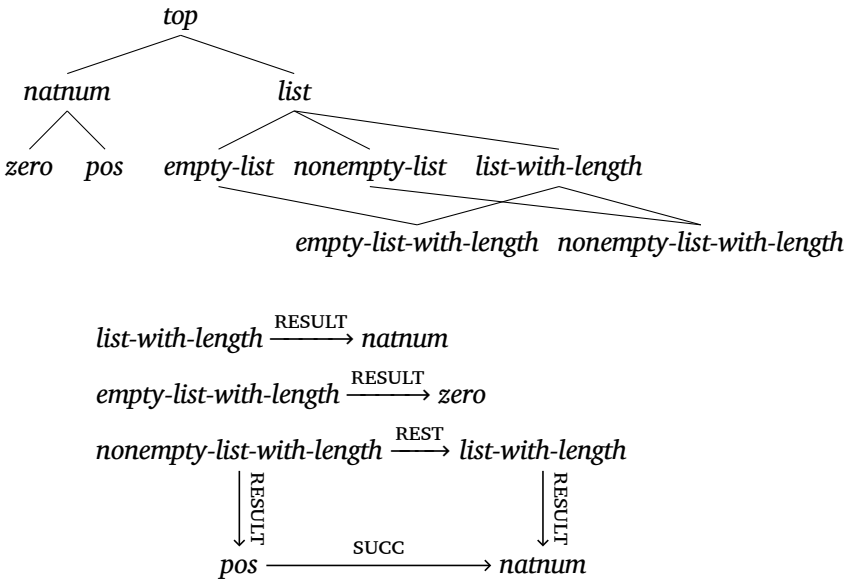


Figure 11: Type system with a function for the length of a list. Constraints for *nonempty-list* and *pos* are not shown; see Figs. 5 and 9. Only a partial constraint is shown for *nonempty-list-with-length*; the full constraint (with FIRST) can be inferred from supertype constraints.

4 TURING COMPLETENESS

In this section, we view type systems as models of computation, and show that they are Turing-complete: any computable function can be encoded in a type system.

A corollary is that checking type systems for consistency is undecidable: given a type hierarchy and partially specified type constraints, there is no algorithm which can always determine whether the constraints can be expanded to give a well-defined type system.

This section can be safely skipped by readers primarily interested in using computation types in a grammar – the linguistic examples in §5 are much more restricted than what is possible in the general case.

In §4.1, we first present a direct way to encode a Turing machine as a type system. In the subsequent sections, we prove two theorems: in §4.2, we show that one-feature type systems are computationally equivalent to finite-state automata, and in §4.3, we show that two-feature type systems are equivalent to Turing machines.

In the rest of this paper, only partial constraints are given, if the full constraints can be easily inferred (as we saw in Fig. 11).

A deterministic Turing machine is a general model of computation (for an exposition, see: Hopcroft and Ullman 1979). It runs in discrete time steps, and has access to a one-dimensional memory tape with discrete cells, which continue indefinitely in both directions. It only reads one cell at a time. At each time step, based on the machine's current state, the machine chooses a symbol to write in the current cell, and chooses whether to move left or right along the tape.

More formally, a Turing machine can be defined by: a finite set of symbols Σ (including a blank symbol $0 \in \Sigma$), a finite set of states Q (including a start state $q_0 \in Q$), and a transition function $\delta : Q \times \Sigma \to \Sigma \times \{\text{left}, \text{right}\} \times (Q \cup \{\text{halt}\})$.[10] The initial tape can only have a finite number of cells that are not blank. Given the initial tape and state, running the machine until it halts is deterministic: at each time step, δ determines the tape and state at the next step.

Fig. 12 shows how a Turing machine can be implemented as a type system, in a relatively direct way. The tape can be represented by two strings, one for the tape to the left of the machine head, and one for the tape to the right. The *string* subhierarchy works similarly to lists and Peano numerals, using a recursive feature REST, but it also introduces a subtype for each symbol. The Turing machine, of type *machine*, can then be represented using two features LEFT and RIGHT, for the two halves of the tape. Beyond the end of each string, the tape is taken to be blank.

Subtypes of *machine* specify the current symbol, and the current state. The subtype *run* introduces the features NEXT for running the machine one step, FINAL for running the machine until it halts, and WRITE for the symbol to write at this step. If the machine halts after the current time step (subtype *halt*), the final tape is the next tape; or, if the machine continues (subtype *continue*), with the final result is propagated by a re-entrancy. Shifting is encoded using the subtypes *shift-left* and *shift-right*, which manipulate the tape strings using the subtype *string-with-pop*, which pops the first element, as shown in Fig. 14. In the case that the string is empty, the blank symbol is returned.

[10] There are various alternative definitions that are computationally equivalent. For example, we could include "no shift" as an alternative to left/right.

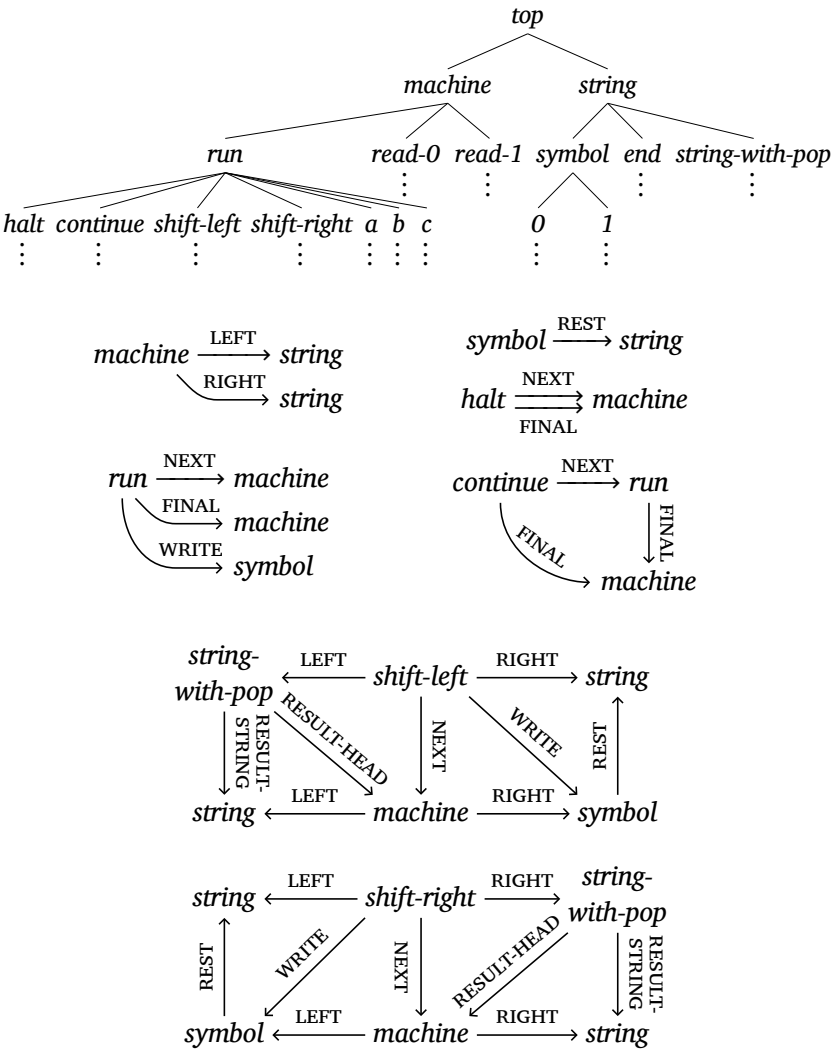


Figure 12: High-level hierarchy for implementing a Turing machine. A feature structure of type *machine* can hold the tape (strings under LEFT and RIGHT), the current symbol (*read-0* or *read-1*), and the current state (*a*, *b*, or *c*). The result of running one step is under NEXT, and the result of running until it halts is under FINAL. The full hierarchy will have exactly one subtype for each combination of state and symbol, as illustrated in Fig. 13, and further subtypes of *string-with-pop*, as shown in Fig. 14.

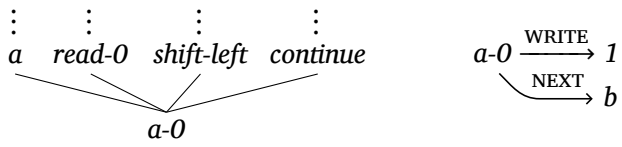


Figure 13: Example of a fully specified type, unifying a state and symbol, and determining the transition.

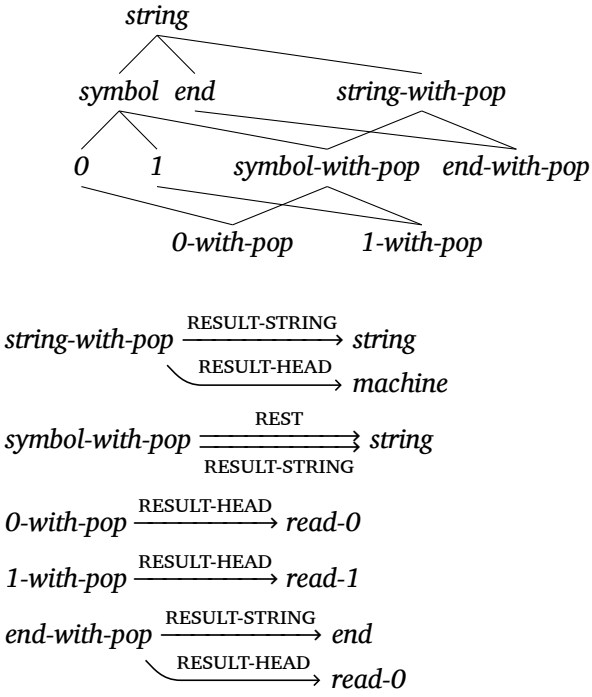


Figure 14: Types for strings (representing one half of the Turing machine tape) and a function for popping the first element, where *0* is the blank symbol (default for an empty string).

Each step of computation is encoded using subtypes. The transition function δ is a function of the current state and symbol, so we need one subtype for each combination of state and symbol, as illustrated in Fig. 13. Each such subtype specifies the symbol to write, whether to shift left or right, whether to halt or continue, and, if continuing, the next state.[11]

Such a type hierarchy is doubly recursive: strings can be arbitrarily long (repeating REST), and there can be arbitrarily many time steps

[11] To give a well-defined type hierarchy, additional GLB types are necessary, for common subtypes of *shift-left/shift-right* and *halt/continue*, and further subtypes of those with a state or symbol. The fully specified state-and-symbol types would inherit from these GLB types rather than directly from the state, symbol, shifting, and halt/continue types. The GLB types ensure that unification is well-defined for any pair of types, but play no role when running a machine from a given state on a given tape. A grammar processing engine like the LKB can add such GLB types automatically.

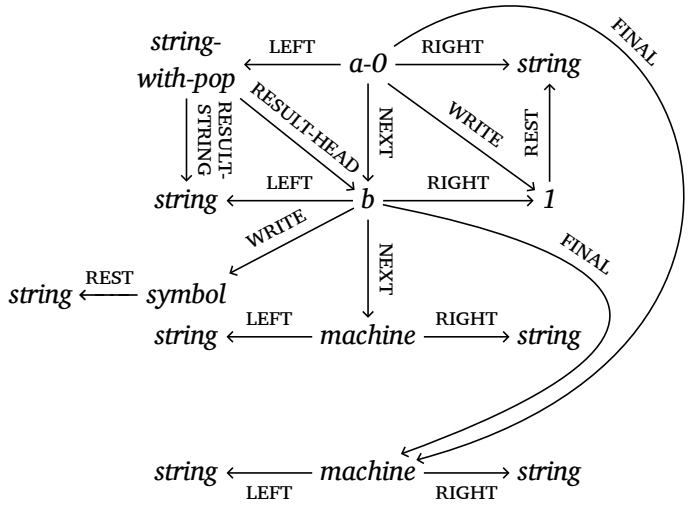


Figure 15: Well-formed constraint, expanding the partial constraint in Fig. 13. This can be inferred from other type constraints. Recursion is limited because the left tape is underspecified, so the next state is not unified with a symbol.

(repeating NEXT). The recursion of these features terminates with the types *end* and *halt*, respectively. However, the recursive types *symbol* and *continue* underspecify what comes next. In fact, there is no type constraint which is fully specified: every maximally specific type has a feature with an underspecified value (subtypes of *machine* underspecify LEFT and RIGHT; subtypes of *string* underspecify RESULT-HEAD).

Because of underspecification, the partial constraints in Figs. 12 to 14 can be easily expanded to well-formed constraints (accumulating information from other type constraints), without having to run the Turing machine. Expanding the partial constraint in Fig. 13 gives the well-formed constraint in Fig. 15. In contrast, unifying a state (with underspecified tape) and a tape (with underspecified state) is Turing-complete: the result is finite iff the Turing machine halts.

As a concrete example, a 3-state 2-symbol "busy beaver" is defined in Fig. 16: this machine maximises the number of non-blank symbols written to the tape, when starting from an blank tape, and ensuring that the machine eventually halts. Unifying the start state *a* with a blank tape gives the result in Fig. 17. Since the state and tape are both fully specified, the computation of the next state triggers the computation of the state after that, and so on.

Turing completeness requires the possibility of unbounded computation. With computation encoded as unification, this means that

Type	Supertypes	WRITE	NEXT
a-0	*a read-0 shift-left continue*	*1*	*b*
a-1	*a read-1 shift-right continue*	*1*	*c*
b-0	*b read-0 shift-right continue*	*1*	*a*
b-1	*b read-1 shift-left continue*	*1*	*b*
c-0	*c read-0 shift-right continue*	*1*	*b*
c-1	*c read-1 shift-right halt*	*1*	-

Figure 16: Defining transitions for a 3-state 2-symbol busy beaver, in the manner of Fig. 13.

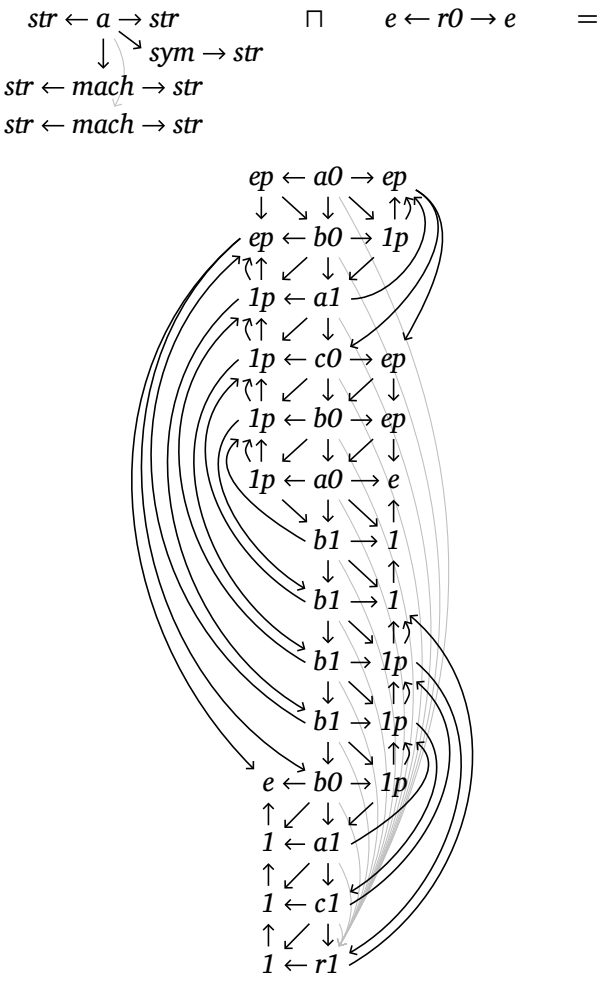


Figure 17: Running the Turing machine given in Fig. 16, by unifying two well-formed feature structures (for the state and tape). To clearly show the pattern of re-entrancies, type names are abbreviated (e.g. *p* for *-with-pop*), feature names are suppressed, and the feature FINAL is shown in grey. (Other features can be unambiguously inferred.) The machine halts after thirteen transitions, and the final tape has six 1s.

(in the general case) we cannot bound the size of the resulting feature structure. In our encoding of Turing machines, the longest path of the form NEXT|…|NEXT is unbounded. Nonetheless, the final tape (if the machine halts) is accessible from the root under the fixed path FINAL, which is re-entrant with the longest path of the form NEXT|…|NEXT.

4.2 *One-Feature Type Systems are Finite-State*

Having seen that Turing machines can be encoded as type systems, it is natural to ask what restrictions on type systems can guarantee reduced computational complexity.

For a type system with only a single feature, no re-entrancies are possible, and a feature structure can be expressed as a string of types.[12] Constraints can only apply "locally" to a substring, which suggests that such systems are of limited complexity. In fact, recognising when two feature structures are unifiable is of equivalent complexity to recognising when a string is accepted by a finite-state automaton (FSA). This can be stated more precisely as the following theorem.

THEOREM 1

i *For any FSA, there is a one-feature type system where unification can determine whether the FSA accepts a string: the string and the start state are each encoded as a feature structure, and their unification is defined iff the string is accepted.*

ii *For any one-feature type system, there is an FSA which recognises when two feature structures are unifiable: they are encoded as a single string with symbols encoding pairs of types, and the string is accepted iff their unification is defined.*

It may be helpful to sketch the proof before presenting it. For simplicity, in part i we assume a deterministic FSA, while in part ii we allow ϵ-transitions, but these formalisms are known to be equivalent (by the subset construction), so this is without loss of generality.

[12] If cycles are allowed, a restricted kind of re-entrancy is possible. A feature structure can be expressed as a string of types followed by a second string that loops back on itself. This complicates the following proof but the result still holds.

For part i, the proof follows the approach in §3.2, but without a RESULT feature – we just need unification success or failure, without any output.[13] The one feature allows us to represent strings, and subtypes are used for both states and symbols. Common subtypes of states and symbols enforce transitions to the next state. Unifying a string and a state triggers the common subtype's constraint, and this propagates until the end of the string.

For part ii, the proof is more involved, since we can have larger constraints, and unification can create a larger structure than either input. However, since the type hierarchy is finite, there is a maximum size of constraint. Each constraint is itself a string of types, which must be "overlaid" at different positions. The maximum constraint size therefore provides a maximum "window size" of previous types that need to be considered, hence only a fixed amount of memory needs to be encoded in the state.

We can define an FSA which proceeds through both feature structures at once. Until it reaches the ends of the structures, it reads the two current types, unifies them, and enforces the previous constraints at this point in the structure. If this is possible, the resulting type is held in memory, the type at the other end of the window is forgotten, and the process continues. At the end of the input structures, ϵ-transitions are used, which continue building the feature structure required by the constraints, but without reading any more input. If this process terminates (which is not guaranteed, as we saw in §2.2), the FSA reaches an accepting state. To control the switch to ϵ-transitions, the state indicates not only the k previous types but also whether we have reached the end of the input.

PROOF OF PART I Let Q, $q_0 \in Q$, $F \subset Q$, Σ, and $\delta : Q \times \Sigma \to Q$ be the states, start state, accepting states, alphabet, and transition function of the deterministic FSA. Let $\Sigma_s = \Sigma \cup \{\mathit{symbol}\}$ and $\Sigma_t = \Sigma_s \cup \{\mathit{top}, \mathit{end}\}$. We define the set of types to be $\Sigma_t \cup (F \times \Sigma_t) \cup ((Q \setminus F) \times \Sigma_s)$. The most general type is *top*, with trivial constraint; *end* is a subtype of *top*, with trivial constraint; *symbol* is a subtype of *top*, introducing the

[13] Alternatively, we can view states as input and symbols as functions (or vice versa!), the single feature as holding the output, and the input string as a sequence of functions to apply to the start state. These different views are simultaneously possible because functions are simply a pattern within the formalism.

one feature, with constraint *symbol* $\rightarrow$ *top*. Each $x \in \Sigma$ is a subtype of *symbol*, with trivial inherited constraint. For each $q \in F$: $(q, \textit{top})$ is a subtype of *top*; $(q, \textit{end})$ is a subtype of $(q, \textit{top})$ and *end*; $(q, \textit{symbol})$ is a subtype of $(q, \textit{top})$ and *symbol*; all with trivial (inherited) constraints. For each $q \in Q \setminus F$, $(q, \textit{symbol})$ is a subtype of *symbol*, with trivial inherited constraint. Each $(q, x) \in Q \times \Sigma$ is a subtype of $(q, \textit{symbol})$ and x, with constraint $(q, x) \rightarrow \delta(q, x)$.

This defines the type system. If $q_0 \in F$, we define the start structure to be $(q_0, \textit{top})$, otherwise we define it to be $(q_0, \textit{symbol}) \rightarrow \textit{top}$. An input string $x_1 \cdots x_n$ can be encoded as the feature structure $x_1 \rightarrow \cdots \rightarrow x_n \rightarrow \textit{end}$. By construction, this is unifiable with the start structure iff the string is accepted by the FSA. □

PROOF OF PART II For any type t, we will denote its constraint as $c(t)$. Any feature structure f is of the form $f_0 \rightarrow \cdots \rightarrow f_n$, for some $n \geq 0$. We will call n the length of f, and denote it as $\text{len}(f)$. For any $i > \text{len}(f)$, let $f_i = \emptyset$.

Let T be the set of types. Let $T_0 = T \cup \{\emptyset\}$. We extend unification over T to unification over T_0, where $\emptyset \sqcap t = t$ for all t. Let k be the maximum length of all constraints.

We define the alphabet to be T_0^2. We define the set of states to be $T_0^k \times \{0, 1\}$, with start state $(\emptyset^k, 0)$. For $a \in T_0^k$, let $u(a) = \bigsqcap_{i=1}^{k} c(a_i)_i$. For state $(a, 0)$ and input symbol (x, y) with $x \neq \emptyset$, let $t = x \sqcap y \sqcap u(a)$. If t is defined, we define one transition with this state and symbol, to state $((t, a_1, \ldots, a_{k-1}), 0)$. For $(a, 0)$ and $(\emptyset, \emptyset)$, if $u(a)$ is defined, we define a transition to $((u(a), a_1, \ldots, a_{k-1}), 1)$. For $(a, 1)$ with $a_1 \neq \emptyset$, if $u(a)$ is defined, we define an ϵ-transition to $((u(a), a_1, \ldots, a_{k-1}), 1)$. Accepting states are of the form $(a, 1)$ with $a_1 = \emptyset$.

This defines the FSA. Let f and g be any two feature structures, with lengths n and m, where $n \geq m$ without loss of generality. They can be jointly encoded as $(f_0, g_0) \cdots (f_m, g_m)(f_{m+1}, \emptyset) \cdots (f_n, \emptyset)(\emptyset, \emptyset)$. By construction, the FSA accepts this string iff f and g are unifiable, with the unique sequence of states from start state to accepting state $(\emptyset^k, 0), (a^{(0)}, b^{(0)}), \ldots, (a^{(l)}, b^{(l)}), (a^{(l+1)}, 1)$ giving the result of unification $a_1^{(0)} \rightarrow \cdots \rightarrow a_1^{(l)}$. □

One-feature type systems and FSAs are of equivalent computational complexity, but the constructions in these proofs do not give a one-to-one mapping between them. In part i, some unifications do

not correspond to running the FSA; and in part ii, some inputs do not correspond to well-formed feature structures.

Nonetheless, the constructions in these proofs can be straightforwardly extended from FSAs to sequential finite-state transducers (FSTs).[14] A sequential FST can be defined in terms of a set of states Q, a start state $q_0 \in Q$, an input alphabet Σ, an output alphabet Γ, a transition function $\delta : Q \times \Sigma \to Q \times \Gamma^*$, and a suffix function $\rho : Q \to \Gamma^*$. This behaves like a deterministic FSA, but also produces an output string in $\Gamma^\star$. The output string is initialised as the empty string, is appended to during each transition (according to the transition function), and is appended to when the machine halts (according to the suffix function). A sequential FST defines a string-to-string function (which may be a partial function, if δ and ρ are partial functions).

Extending the construction in part i, given a sequential FST, we can define additional types for the output alphabet, and an additional RESULT feature to hold the output string (if the output alphabet is distinct from the input alphabet, this second feature is not recursive). For each (q, x) type, the constraint specifies an output substring (according to δ), using a re-entrancy to connect up the output (as we saw in §3.2). For each (q, end) type, the constraint specifies an output string (according to ρ), ending the output with a *end*.

Extending the construction in part ii, given a one-feature type system, we can define the output alphabet to be the set of types. We know that the output string should be $a_1^{(1)} \cdots a_1^{(l)}$. However, the ϵ-transitions must be removed to fit the above definition of a sequential FST. For $(a, 0)$ states, there are no ϵ-transitions, and the transition function can be extended to output a_1. For $(a, 1)$ states, there are only ϵ-transitions, but we can use the powerset construction to determine which ones will eventually reach an accepting state. For those states, we can follow the sequence of states reached by ϵ-transitions (which is deterministic), and define ρ to output the concatenation of a_1 of these states, i.e. $a_1^{(n+1)} \cdots a_1^{(l)}$.

[14] The term "sequential" follows recent scholarship (Lothaire 2005; Lombardy and Sakarovitch 2006; Lambert 2022), replacing the older term "subsequential" which followed Schützenberger (1977) ("sous-séquentiel" in the original French).

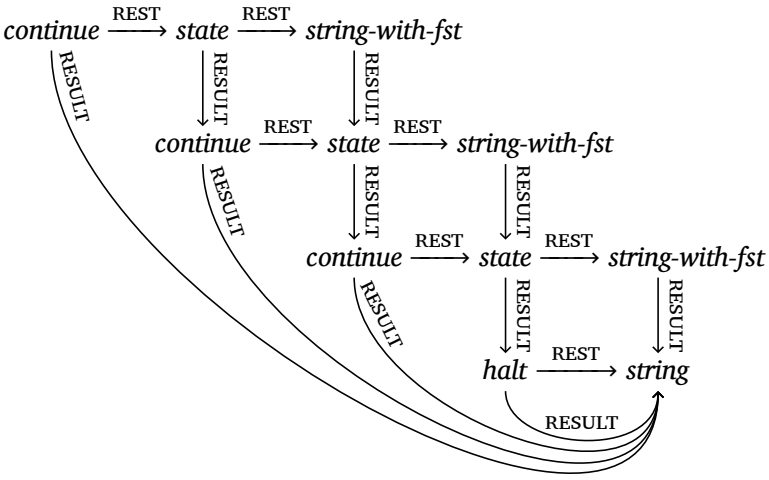


Figure 18: An illustration of the re-entrancies for a recursive transduction machine halting after three time steps. In an actual run of a machine, all nodes except the *continue* and *halt* nodes would have more specific types.

4.3 *Two-Feature Type Systems are Turing-Complete*

However, constraining a type system to have only two features does not in fact constrain complexity at all. In this section, we show that two features are sufficient for Turing completeness, which can be stated as the following theorem. Unlike Theorem 1, this theorem does not have two parts, because no model of computation is more powerful than a Turing machine (by the Church-Turing thesis).

THEOREM 2 *For any Turing machine, there is a two-feature type system where unification can determine whether the Turing machine halts on a given input: the input tape and the start state are each encoded as a feature structure, and their unification is defined iff the machine halts.*

We sketch the proof before presenting it. The intuition is that we need one feature for the data, and one feature for computation. Any data can be linearised (so one feature is sufficient for the data), and one step of computation can apply an arbitrary FST (as we saw in §4.2). While applying a fixed number of FSTs can be represented as a single FST, unbounded recursion is more powerful.

The construction is illustrated in Fig. 18, using the type system in Fig. 19. Compared to the direct construction in §4.1: the feature REST combines the features REST, LEFT, and RIGHT; the feature RESULT combines the features RESULT-STRING, RESULT-HEAD, NEXT,

and FINAL; and the feature WRITE is no longer needed, as writing is determined by the choice of FST.

The *machine* type and its subtypes *continue* and *halt* are only needed to propagate the final string. The type system would still be Turing-complete without them, but the feature path to the final string would have variable length.

Recursively applying FSTs is sufficient for Turing completeness, because we can simulate tape operations using FSTs. We can represent a two-ended tape with a single string by interleaving the elements: odd elements of the string hold the left tape, and even elements hold the right tape. This means "logically adjacent" symbols are exactly two symbols away. A Turing machine transition requires pushing a symbol to one side of the tape and popping a symbol from the other side. This can be performed by an FST whose states encode two symbols and which side of the tape it is on. After reading the first two symbols, one symbol is popped to the output, while the other is held in memory along with the symbol to be pushed. On the "push" side of the tape, one symbol is output from memory and forgotten, while the input symbol is saved to memory; on the "copy" side of the tape, the input symbol is copied to the output, while the memory is left unchanged.

PROOF OF THEOREM 2 We define a *recursive transduction machine* in terms of: a set of states Q, a start state $q_0 \in Q$, an alphabet Σ, a set S of sequential FSTs with input and output alphabet Σ, and a transition function $\delta : Q \times \Sigma \to S \times (Q \cup \{\text{halt}\})$. Given an initial string, the machine operates deterministically. Given the current state and the first element of the current string, the transition function gives an FST to apply to the rest of the string, which becomes the next string, and also gives the next state. This continues until the halt state is reached.

Such a machine can be implemented as a two-feature type system, as shown in Figs. 19 and 20. The type *machine* has two features: REST holds the current string and (if continuing) the current state, while RESULT holds the final string if the machine eventually halts. The subtypes *halt* and *continue* ensure the final string is propagated. As a corollary of Theorem 1, any sequential FST can be implemented in a type system with two features; we use REST to recursively link adjacent symbols, and RESULT to link the input to the output. The transition function is implemented with subtype constraints; for each combina-

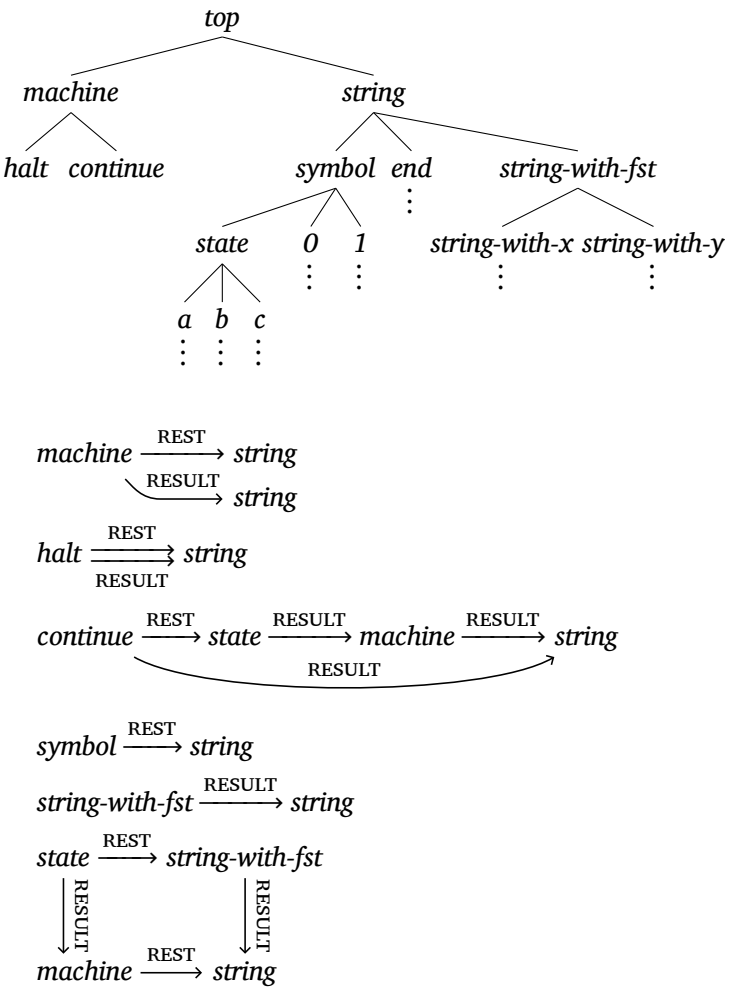


Figure 19: High-level hierarchy for a two-feature machine. The immediate subtypes of *string-with-fst* and *state* can be seen as functions on *string* and *symbol*, respectively. Each state type will have one subtype for each symbol, with examples shown in Fig. 20. Each FST type will have one subtype for each symbol and also for *symbol* and *end*. Three additional types would be necessary to ensure feature appropriateness (one for each feature, and one for combining both).

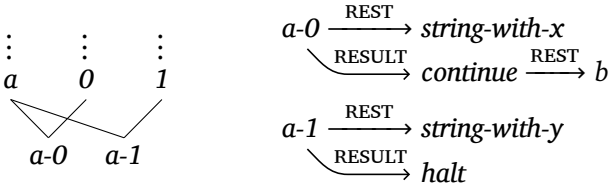


Figure 20: Examples of unifying states and symbols. Each constraint specifies the FST to apply to the string, and the next state.

tion of state and symbol, we define a unique common subtype, whose constraint specifies: an FST type on REST (hence applying the FST to the rest of the string); *halt* or *continue* on RESULT; and, if continuing, a state on RESULT|REST (which is re-entrant with REST|RESULT).

It remains to be shown that a recursive transduction machine can simulate a Turing machine. The tape can be encoded as a string by alternately encoding one symbol from each side. Writing to the tape and shifting left/right must be simulated with sequential FSTs.

We define an FST with states $(\Sigma^2 \times \{P, C\}) \cup (\Sigma \times \{L, R, R'\})$. We define transitions as: $(x, y, P), z \mapsto (y, z, C), x$; $(x, y, C), z \mapsto (x, y, P), z$; $(x, L), y \mapsto (x, y, C), \epsilon$; $(x, R), y \mapsto (x, R'), y$; $(x, R'), y \mapsto (x, y, C), \epsilon$. We define suffixes as: $(x, y, P) \mapsto \text{trim}(x0y)$; $(x, y, C) \mapsto \text{trim}(0x0y)$; $(x, L) \mapsto 0\,\text{trim}(x)$; $(x, R) \mapsto 0\,\text{trim}(0x)$; $(x, R') \mapsto \text{trim}(0x)$; where 0 is the blank symbol and $\text{trim} : \Sigma^\star \to \Sigma^\star$ removes trailing blanks. Starting in state (x, L) or (x, R), this FST writes x and shifts left or right, respectively.

□

An immediate corollary of Theorem 2 is that any computable function can be encoded in a two-feature type system, where possible inputs are encoded as feature structures with one feature, and where the function is encoded as a type, with the output under the second feature. Unification is defined iff the function is defined on the input.

As an example, a 3-state 2-symbol busy beaver is shown in Fig. 21.

Avenues for future work would be to identify further classes of type system with different levels of complexity, and to find methods for establishing the complexity of a given type system. The practical applications in §5 use two-feature type systems that are only finite-state in complexity, with no recursion of the RESULT feature.

TODO! Sketch time and space... explain lack of separation between data and program, like lambda calculus (Vanoni 2022)... Mention subregular? ... Sketch context-free... parse chart...

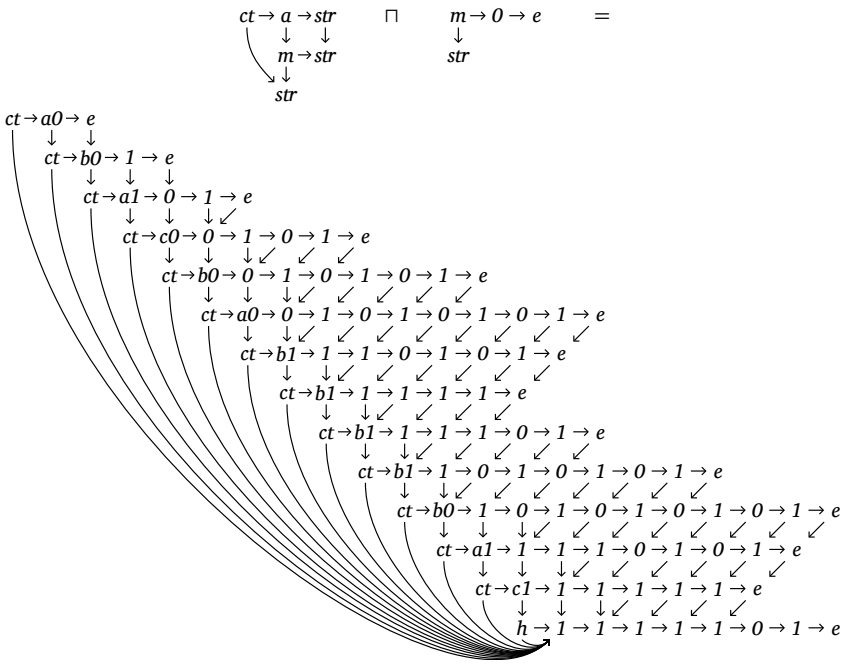


Figure 21: The same Turing machine as in Fig. 17, but encoded with only two features, as in Figs. 18 to 20. REST is to the right; RESULT is down. FST types are suppressed: except in the bottom row, all *0*, *1*, and *e* nodes should have a more specific type, with one of the FST states in the proof of Theorem 2.

5 LINGUISTIC APPLICATIONS

The aim of this section is to make computation subtyping easier to apply in a practical grammar. We don't need the full expressive power of Turing completeness, but we also don't need to explicitly restrict the formalism. As explained by Bender and Emerson (2021), distinguishing formalism from theory allows the maintenance of software systems that target the formalism, and a powerful formalism allows a range of theories to be expressed.

A useful formalism allows us to state linguistic generalisations clearly and easily. In particular, relational constraints are widely used by HPSG grammarians, but are not available in the DELPH-IN Joint Reference Formalism, as discussed by Meurers *et al.* (2003) and Melnik (2007). We will first introduce the syntactic framework in §5.1, and then explain how to mimic relational constraints in §5.2. This allows a wider range of theoretical proposals to be directly implemented, closing the gap with other grammar processing engines such as TRALE.

We will give several examples of relational constraints, covering logical operations in §5.3 and list operations in §5.4. Finally, we will present a mechanism for implementing nondeterministic relational constraints in §5.5. In all of these cases, defining the type system can be done once, and the resulting types can be easily used by a grammarian. No changes are required to the processing engines.

For the rest of the paper, we will switch to AVM notation of feature structures, which is more usual in the linguistics literature. Compared to the graph notation that we have used so far, re-entrancies are less obvious, but multiple features can be stated more succinctly. Lists can be indicated with ⟨⟩ notation, even though they are not formally special (as we saw in §3.2.2).

Formal Framework 5.1

A typed feature structure grammar is defined by a type system, a set of lexical entries (well-formed feature structures, with a distinguished feature (PHON) holding the input token), a set of rules (well-formed feature structures, with a distinguished feature (DTRS) holding a list of feature structures for the daughters), and a set of root conditions (well-formed feature structures).[15]

The rules are manipulated using an expanded notion of unification, where a structure F can have a substructure at path α unified with a structure G (more precisely, we append α to the beginning of all paths in G, and then unify the resulting structure with F).

A grammar accepts strings via phrase structure trees. A string is accepted if there exist a tree over the string, a root condition, a rule for each non-leaf node, and a lexical entry for each leaf node (with the token matching PHON), such that the feature structures can all be unified: for each non-leaf node, the elements of the rule's DTRS are unified with the daughter feature structures (themselves either rules or lexical entries), and the root condition is unified with the root feature structure (either a rule or lexical entry).

[15] This simple picture may be more complicated in practice. A grammar might let a lexical entry match many tokens (e.g. using a regular expression), and might use lexical rules, which are unary rules that can manipulate the PHON feature. Such details will not be discussed further in this paper.

Figure 22: Example use of a wrapper type, for logical “and”.

$$\begin{bmatrix} \textit{my-phrase} \\ \textsc{my-path} & \left[\textsc{and} \left\langle \boxed{1}, \boxed{2} \right\rangle\right] \\ \textsc{dtrs} & \left\langle \left[\textsc{my-path}\ \boxed{1}\right], \left[\textsc{my-path}\ \boxed{2}\right] \right\rangle \end{bmatrix}$$

5.2 *Wrapper Types: Relational Constraints without Relational Constraints*

In this section, we present a framework for mimicking relational constraints. The intended use case is allowing a feature value on the mother to be defined as a function of feature values on the daughters.

We saw in §3 that functions can be implemented as subtypes, and we saw in §4 that any computable function can be implemented in this way. However, directly programming with such subtypes is awkward, particularly for functions with multiple arguments: each daughter feature needs to be subtyped, and the RESULT features need appropriate re-entrancies.

Wrapper types allow a more straightforward way to state constraints, effectively providing an intuitive “programming interface” for calling computation types. This is illustrated in Fig. 22, where a boolean feature on the mother is specified as the logical “and” of the daughters’ values. The boolean value is not directly held on MY-PATH, but rather on MY-PATH|BOOL; the value of MY-PATH is a wrapper type, in particular *bool-wrapper*. Once the type constraints have been enforced, the value of MY-PATH|BOOL on the mother will be the logical “and” of the values on the daughters. The extra BOOL feature is necessary to cleanly separate the output of the function from the rest of the computation.[16]

In §5.2.1, we suggest best practices for working with wrapper types. There is no formal status to these best practices, since the formalism is unchanged from §2. Instead, they are best seen as a software design pattern for grammar engineering: a general reusable approach to implementing relational constraints. Wrapper types can be defined once, and then used in many rules.

[16] If cycles are allowed, the wrapping feature could be avoided. However, this would introduce the complication that the type on the mother is a subtype, rather than a “clean” data type (see §5.2.2).

In §5.2.2, we then warn of possible pitfalls if wrapper types are used outside of the intended use case. Finally, in §5.2.3, we discuss the notion of "computation history" when composing wrapper types, and contrast two ways that wrapper types could be implemented.

Suggested Best Practices 5.2.1

There should be a *with-computation* type which introduces the RESULT feature. Using the same feature for all functions provides clarity of intention. Using a supertype for the feature is necessary for feature appropriateness, and provides further clarity of intention.

All types inheriting from *with-computation* are called computation types, and the value of RESULT should hold the output of the function being computed. A computation type should also inherit from another type, which is the input to the function. The input and output types are called data types.

Computation types should be named *x-with-y*, where *x* is the name of the input type,[17] and *y* describes the function. Defining a function requires one subtype for each type in the input hierarchy, and all of these should be named with the same *y*.[18]

There should be a *wrapper* type which has no features. This type is not intended to be used directly, but using it as a supertype for all wrapper types provides clarity of intention.

Immediate subtypes of *wrapper* should be named *x-wrapper*, and should introduce the feature X to hold data of type *x*. A supertype for the X feature is necessary for feature appropriateness, and also provides clarity of intention.

Each subtype of an *x-wrapper* type should introduce a feature which invokes one or more computation types, and should make the result of the computation re-entrant with X.

---

[17] Or a modified version of the name, such as in the case of a list of elements of a specific type, as we will see in §5.4.

[18] Using an inference engine that can automatically infer GLB types, it is only strictly necessary to define types for the most general type and all the most specific types, as the other types can be automatically inferred. For some functions, constraints on intermediate types are useful (as can be seen in Fig. 3), and explicitly naming GLB types can be helpful when viewing inferred feature structures.

The names of the wrapper features are more important than the names of the wrapper types, because the features are how the wrapper types are expected to be invoked. By assigning a unique feature to each wrapper type, enforcing feature appropriateness means that invoking the computation can be done simply by using the feature, as illustrated by the feature AND in Fig. 22. The feature triggers the wrapper type, which in turn triggers the computation type(s). The name of the feature is effectively the "programming interface" of the computation.

5.2.2 "Clean" Data and Pitfalls to Avoid

The intended use of wrapper types is as described above: some feature value on the mother is a function of some values on the daughters. In this case, the value on the mother is a "clean" data type, in the sense that it is not subtyped to a computation type. However, the daughter types are "destructively" modified: they are unified with computation types, and so will not be clean data types. If these destructively modified types play no other role in the derivation, this is not a problem. However, this does mean that care must be taken if applying wrapper types in some other potential applications of relational constraints.

When applying a wrapper type, information "flows" from the input to the output.[19] For example, in Fig. 22, specifying the value of MY-PATH|BOOL on either daughter to be *false* would force the value of MY-PATH|BOOL on the mother to also be *false*. However, the converse is not true: specifying *true* on the mother is compatible with un underspecified *bool* on both daughters (*true* would be the only logical possibility, and specifying *false* on either daughter would cause unification failure, but the values can nonetheless remain underspecified). This directionality (from input to output) is an important difference compared to true relational constraints.[20]

In the intended use, information flows from daughter to mother. If wrapper types are always applied in this way, the mother's data

[19] This follows from the nature of the formalism and how computation types are defined. For any feature path $\alpha\beta$, enforcing a more specific type at α can force a more specific type at $\alpha\beta$, but not vice versa. In particular, specifying a type at α can force a more specific type at α|RESULT, but not vice versa.

[20] If cycles are allowed, "bidirectional" computation is possible, but this would mean that there is no clean output.

is always clean, and can serve as input to another rule. This allows information to flow through the derivation, from lexical entries to the root of the phrase structure tree.

One possible pitfall is specifying a value as input to more than one wrapper type. The input value must be subtyped, and it's not possible to subtype it in multiple ways at once.[21]

Another possible pitfall is applying wrapper types in ways that don't pass information from daughter to mother. For example, in a "lexical threading" analysis of long-distance dependencies (Bouma *et al.* 2001), the SLASH value in a lexical entry is specified as a function of the SLASH values on its SUBJ and COMPS lists. This means that the SUBJ and COMPS lists are not clean, which would cause problems if these lists are constrained in other ways. To bring lexical threading in line with the intended use of wrapper types, append operations would have to be delayed until later in the derivation.[22]

Computation History 5.2.3

In the example in Fig. 22, two wrapper types are given as input to a wrapper type. This allows wrappers to be composed within a single rule, as illustrated in Fig. 23.

However, a downside is that "computation history" is exposed beyond the current computation: a wrapper type can hold not only the input and output data, but also all previous computations that were used to produce the input. For example, in Fig. 23, the substructure at MY-PATH contains not only the NOT operation, but also the AND operation. When using wrapper types in a grammar, they will often be composed recursively through a series of phrase structure rules. If the

[21] It would be possible to define a computation type that outputs multiple "copies" of the input, so that each copy could be passed to a different wrapper. However, applying such a type would be much more cumbersome than the example in Fig. 22. For grammar maintainability, we would not recommend this.

[22] For example, this could be done with a bookkeeping feature that holds the part of the complement/subject SLASH list that should be appended to the head's SLASH list. This feature has its value specified in the lexical entry, following the lexical threading analysis, but the append operation is delayed until the *head-comp* or *head-subj* rule is applied. This is conceptually the inverse of the TO-BIND feature proposed by Pollard and Sag (1994).

Figure 23: Wrappers can be composed, if inputs are wrappers.

$$\begin{bmatrix} \textit{my-complex-phrase} \\ \textsc{my-path} & \left[\textsc{not}\ \left[\textsc{and}\ \left\langle \boxed{1}, \boxed{2} \right\rangle\right]\right] \\ \textsc{dtrs} & \left\langle \left[\textsc{my-path}\ \boxed{1}\right], \left[\textsc{my-path}\ \boxed{2}\right] \right\rangle \end{bmatrix}$$

Figure 24: Computation history must be cut off, if inputs are data types.

$$\begin{bmatrix} \textit{my-alternative-phrase} \\ \textsc{my-path} & \left[\textsc{alternative-and}\ \left\langle \boxed{1}, \boxed{2} \right\rangle\right] \\ \textsc{dtrs} & \left\langle \left[\textsc{my-path}\ \left[\textsc{bool}\ \boxed{1}\right]\right], \left[\textsc{my-path}\ \left[\textsc{bool}\ \boxed{2}\right]\right] \right\rangle \end{bmatrix}$$

Figure 25: Computation history can be cut off, if inputs are wrappers.

$$\begin{bmatrix} \textit{my-history-trimming-phrase} \\ \textsc{my-path} & \left[\textsc{and}\ \left\langle \left[\textsc{bool}\ \boxed{1}\right], \left[\textsc{bool}\ \boxed{2}\right] \right\rangle\right] \\ \textsc{dtrs} & \left\langle \left[\textsc{my-path}\ \left[\textsc{bool}\ \boxed{1}\right]\right], \left[\textsc{my-path}\ \left[\textsc{bool}\ \boxed{2}\right]\right] \right\rangle \end{bmatrix}$$

full computation history is maintained in the feature structures, this can lead to a high processing cost.[23]

To avoid this processing cost, computation history should be cut off. This can be forced, by defining wrapper types to use data types as input, as illustrated in Fig. 24. However, this makes it impossible to compose wrappers within one rule. In contrast, when using wrapper types as input, it is still possible to cut off computation history, but this must be done explicitly, as illustrated in Fig. 25.

In the rest of this paper, we will assume wrapper types as input, since this allows flexible composition (as in Fig. 23), but cutting off computation history is still possible (as in Fig. 25).

5.3 *Boolean Operations*

We saw logical negation and logical "and" in §3.1, and we can now define wrapper types for them, as shown in Fig. 26. Negation is a unary operation, and so the input under NEG is a single *bool-wrapper*.

[23] Features that recursively maintain history without recursively constraining grammaticality (such as DTRS) are typically suppressed during parsing after each unification, to reduce computational cost. Wrapper features are recursive but can constrain grammaticality, making them difficult to control in the same way.

$$\left[\begin{array}{l}\textit{not-bool}\\ \textsc{bool}\ \boxed{1}\\ \textsc{not}\ \left[\textsc{bool}\ \left[\begin{array}{l}\textit{bool-with-neg}\\ \textsc{result}\ \boxed{1}\end{array}\right]\right]\end{array}\right]$$

$$\left[\begin{array}{l}\textit{and-bool}\\ \textsc{bool}\ \boxed{1}\\ \textsc{and}\ \left\langle\left[\textsc{bool}\ \left[\begin{array}{l}\textit{bool-with-bool-pair-1st}\\ \textsc{result}\ \boxed{2}\end{array}\right]\right],\left[\textsc{bool}\ \left[\begin{array}{l}\textit{bool-with-bool-pair-2nd}\\ \textsc{result}\ \boxed{2}\left[\begin{array}{l}\textit{bool-pair-with-and}\\ \textsc{result}\ \boxed{1}\end{array}\right]\end{array}\right]\right]\right\rangle\end{array}\right]$$

Figure 26: Wrapper types for negation and logical "and". Each type uses a unique feature, and invokes one or more computation types.

Logical "and" is a binary operation, and the input under AND is a list of two *bool-wrapper*s.[24] In both cases, computation types are invoked, with the result re-entrant with BOOL.

However, Fig. 26 departs from Fig. 3 by encapsulating the currying of the function, as explained in §5.3.1. This makes it easier to define functions of multiple arguments.

In §5.3.2, we will apply boolean wrapper types to modelling coordination, where agreement features of a coordinated phrase are often a function of the features of the conjuncts.

Encapsulated Currying 5.3.1

Defining functions with multiple arguments requires currying, and as we saw in Fig. 3, this can be intricate. However, we can separate the definition of the Cartesian product from the definition of the function.

The hierarchy in Fig. 27 defines an ordered pair of booleans. Using these types as input, it is straightforward to define binary operations (such as logical "and", "or", or implication). This would be similar to the function hierarchy in Fig. 3, but without being subtypes of *bool*. For example, the type *true-true-with-and* would inherit from *true-true*, *true-bool-with-and*, and *bool-true-with-and*, and its constraint would specify $\left[\textsc{result}\ \textit{true}\right]$.

[24] An alternative interface would be to use two features, which might be more intuitive for some other operations.

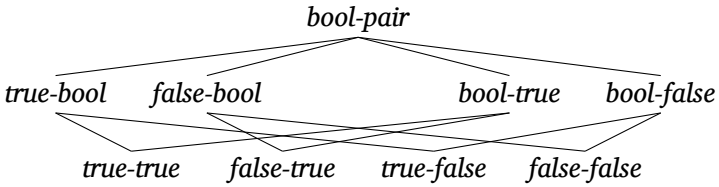


Figure 27: Type hierarchy for an ordered pair of bools.

$$\begin{bmatrix} \textit{bool-with-bool-pair-1st} \\ \textsc{result}\ \textit{bool-pair} \end{bmatrix} \quad \begin{bmatrix} \textit{bool-with-bool-pair-2nd} \\ \textsc{result}\ \textit{bool-pair} \end{bmatrix}$$

$$\begin{bmatrix} \textit{true-with-bool-pair-1st} \\ \textsc{result}\ \textit{true-bool} \end{bmatrix} \quad \begin{bmatrix} \textit{true-with-bool-pair-2nd} \\ \textsc{result}\ \textit{bool-true} \end{bmatrix}$$

$$\begin{bmatrix} \textit{false-with-bool-pair-1st} \\ \textsc{result}\ \textit{false-bool} \end{bmatrix} \quad \begin{bmatrix} \textit{false-with-bool-pair-2nd} \\ \textsc{result}\ \textit{bool-false} \end{bmatrix}$$

Figure 28: Type constraints for encapsulated currying of boolean operations.

In order to use the *bool-pair* type, we need computation types to convert from a pair of *bool* types to a *bool-pair*. Fig. 28 gives type constraints for *bool-with-bool-pair-1st* and *bool-with-bool-pair-2nd*, which do exactly this. These types can be straightforwardly applied, as seen in Fig. 26. Currying is encapsulated, in the sense that these types can be reused by other binary functions, such as logical "or" and logical implication.

Compared to Fig. 3, Fig. 26 still has two layers of computation (two RESULT features in a row), but it maintains symmetry between the two arguments.

This general approach can be generalised to other domains, and we will see an example in §5.3.2, for grammatical number. Ternary and higher arity functions can also take similar approach. For an input hierarchy with n types, a k-ary function can be defined using a product hierarchy with n^k types. Flexible arity, such as logical "and" for an arbitrary list of values, can be defined recursively as explained in §5.4.

5.3.2 Application: Coordination

TODO! expand

Person, number, gender (etc.) are often some function of the values of the conjuncts. Grammar Matrix has implemented this with rules, but requires one rule for each combination of values on the daughters (Drellishak and Bender 2005). Instead, single rule that enforces computation types.

Aguila-Multner and Crysmann (2018) following prescriptive grammar of French. Can simplify analysis, using bools and a logical operation (masculine feature with "or" or feminine feature with "and"), and using wrapper types to make the grammar more maintainable.

singular, dual, plural – "addition" operation. first, second, third – "minimum" operation.

List Operations 5.4

Lists are a useful data structure, allowing a flexible number of objects to be collected together. Lists can be used to define functions with flexible arity, as discussed in §5.4.1.

Lists are often used in linguistic representations, for both syntactic information (such as a list of complements) and semantic information (such as a list of predications). It is often useful to be able to append two lists, combining them into a single list. However, this cannot be done by unifying the lists, and requires another mechanism such as relational constraints. In fact, Müller (2015) says that "The relational constraint that is used most often in HPSG is append".

The ubiquity of list appends and the impossibility of appending by direct unification has led to multiple proposals for how to implement the append operation. We will discuss appending with difference lists in §5.4.2, with junk slots in §5.4.3, and finally with computation types in §5.4.4. We will close this section with two applications of list appends, multiple extraction in §5.4.5 and valence changes in §5.4.6.

Functions with Flexible Arity 5.4.1

In some cases we may want to define a function with flexible arity, for example recursively applying a binary operation to all arguments. This includes taking the logical "and" of a list of booleans, summing a list of natural numbers, or appending a list of lists.

To do this, we can define a computation type for a list. Constraints will propagate along the list, as we saw in §3.2.2. For a nonempty list, we return the result of applying the binary operation to the current element of the list and the result of the rest of the list. These constraints propagate along the whole list, recursively applying computation types to all elements.

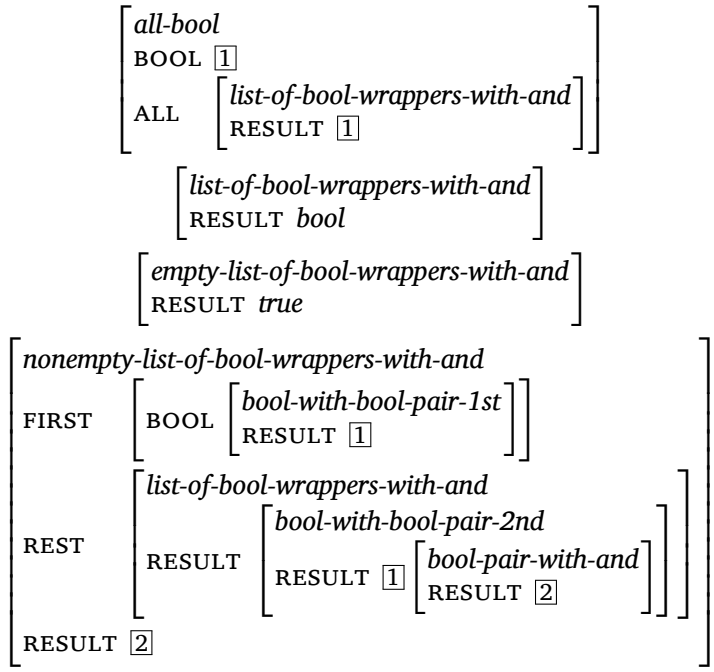


Figure 29: A wrapper type for logical "and" taking any number of booleans as input, along with its associated computation types. The computation is similar to Fig. 26, but recursively applied along the list.

For an empty list, we need to return some "default" value. This is typically the identity for the operation, such as: zero for addition, one for multiplication, true for logical "and", false for logical "or", and the empty list for appending.

A wrapper type invoking such a computation type accepts a list of any size as input, as illustrated in Fig. 29 for logical "and".

5.4.2 Appending with Difference Lists

The append operation takes two lists and combines them: the result is a single list that begins with all the elements of the first list, and then continues with all the elements of the second list. However, this operation cannot be implemented as a simple unification. The *empty-list* at the end of the first list cannot be unified with the second list, unless the second list is empty. So, we can see that fully specified lists cannot be directly appended. We will refer to a list feature structure ending with *empty-list* as a "closed" list. In contrast, an underspecified list that ending with *list* will be referred to as an "open" list.

Figure 30: Appending with difference lists.

Difference lists augment open lists with a bookkeeping feature LAST to hold the open end of the list. This allows appending, as illustrated in Fig. 30, where the first difference list is the append of the other two.[25] Formally, the type constraint for *diff-list* simply specifies that both LIST and LAST are of type *list*. However, this is done with the expectation that LAST should be re-entrant with LIST|REST^n, for some $n \geq 0$.

A difference list can be seen both as a data type and as a wrapper type for a list (recall that there is no formal distinction, just a convention, so both views are simultaneously possible). Viewed as data type, a difference list is defective, in the sense that the list is not intended to be fully specified. Viewed as a wrapper type, a difference list is a parametrised function, in the sense that the type constraint requires further specification to define a function: given an open list under LIST with an appropriate re-entrancy, the feature structure encodes a function that takes an input list (under LAST) and gives an output list (under LIST) with additional elements appended to the start.

The downside of difference lists is that they are awkward to use. Appending two lists, as in Fig. 30, is not intuitive. Furthermore, because difference lists exploit underspecification, it is not possible to check their length without making further appends impossible. For example, consider a phrasal rule which should check the first element of a difference list on a daughter, and then pass the rest of the list to the mother. Intuitively, this should only be possible if there is at least one element between LIST and LAST. However, an empty difference list is simply one where LIST and LAST are re-entrant. Specifying that LIST should be of type *nonempty-list* is compatible with this re-entrancy. Further specifying that LAST should be of type *empty-list* would enforce that there is at least one element between LIST and LAST, but this closes the list, blocking further appends. The intuitive constraint that there is at least one element between LIST and LAST cannot be ex-

[25] For further exposition and history, see: Geske and Goltz 2007.

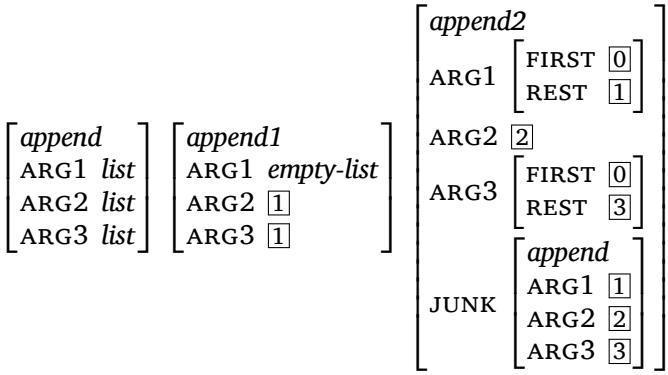


Figure 31: Type constraints for appending with junk slots, also requiring the disjunctive constraint that *append* implies either *append1* or *append2*.

pressed as a feature structure, since the necessary re-entrancy involves a path of variable length.[26]

5.4.3 Appending with Junk Slots

As mentioned in §2.1, Aït-Kaci (1984) showed that relational constraints can be implemented using so-called "junk slots" (for an exposition of junk slots for list appends, see: Götz and Meurers 1996). In this section, we contrast this against our proposed approach.

Aït-Kaci worked within a more general framework than that presented in §2, allowing type constraints to be arbitrary logical formulae, rather than feature structures (for more detail, see: Carpenter 1992, §15). To implement the append operation, Aït-Kaci used a disjunctive constraint. Disjunction is not part of the formalism considered in this paper, and the closest analogue is to introduce a subtype for each disjunct. Feature structures are given in Fig. 31, where *append1* and *append2* are subtypes of *append*, with the intention that ARG1 and ARG2 are appended to give ARG3. In contrast to computation types, this does not introduce any subtypes of *list*.

The two subtypes *append1* and *append2* are mutually exclusive, enforcing that ARG1 is empty or nonempty, respectively. If we can stipulate the logical formula $append \implies append1 \vee append2$, then the

[26] It is possible to define a subtype which results in unification failure if there is no such element (subtyping the LIST|REST list to have a feature re-entrant with LAST, which creates a cycle precisely in such a case). However, the use of such a type would be an acknowledgement that computation by subtyping is necessary for full functionality, and the subtyping approach in §5.4.4 is more intuitive.

JUNK feature in *append2* will recursively apply *append*, propagating through the whole of ARG1.

A related mechanism is sort resolution: this requires every type in a well-formed feature structure to be maximally specific in the type hierarchy. With sort resolution, merely specifying *append* is not well-formed. If *append1* and *append2* are the only subtypes of *append*, one of the two must apply.

However, in our formalism, there is no way to stipulate a logical proposition, and feature structures are not required to be sort-resolved. Using the *append* type in Fig. 31 will therefore not have the desired effect. The type constraint for *append* is well-formed, and there is no mechanism that would force its subtypes' constraints to apply. This means that, no matter what lists are unified with ARG1 and ARG2, ARG3 will remain an underspecified list.

However, allowing disjunctive constraints or requiring sort resolution would be a nontrivial change to the formalism. In either case, it would mean that unification no longer produces a unique result.

Appending with Subtyping 5.4.4

A wrapper type can be used to append lists, with two steps of computation: we first convert the lists to difference lists, then append the difference lists. Both steps involve recursive constraints propagated along a list, and type constraints are given in Fig. 32.

The *list-with-diff-list* type and its subtypes together define a function from lists to difference lists, where the result contains the same elements as the original list. The constraint for an empty list returns an empty difference list. The constraint for a nonempty list includes the current element and propagates the open end of the list.

The *list-of-list-wrappers-with-append* type and its subtypes together define a function from a list of list wrappers to the result of appending the wrapped lists. This is a function with flexible arity, as explained in §5.4.1. The constraint for an empty list returns an empty list. The constraint for a nonempty list converts the current wrapped list to a difference list, and identifies its open end with the result of appending the rest of the wrapped lists.

Although the constraints in Fig. 32 are intricate, the *append-list* type can be straightforwardly invoked using the APPEND feature. We will see examples in the following two sections (§5.4.5 and §5.4.6).

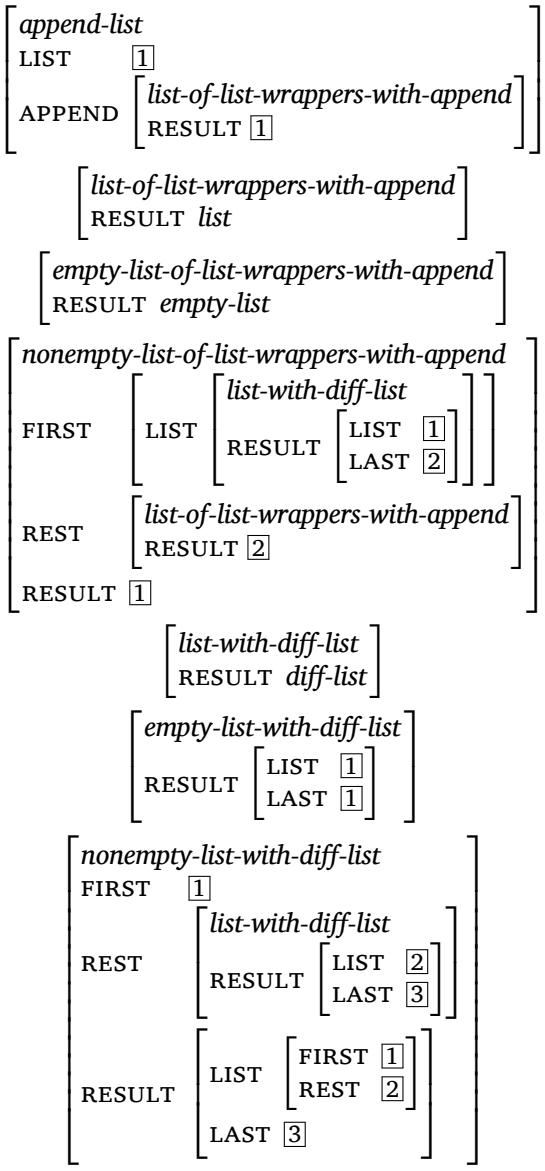


Figure 32: A wrapper type for appending lists, and its associated computation types. The *list-of-list-wrappers-with-append* type allows the input to be any number of lists: it recursively appends each list to the next. Appending lists requires the *list-with-diff-list* type, which recursively creates a new open list and propagates the open end, as illustrated in Fig. 33. To append two lists, the first list is converted to a difference list, and its open end is identified with the second list, as seen in the constraint for *nonempty-list-of-list-wrappers-with-append*.

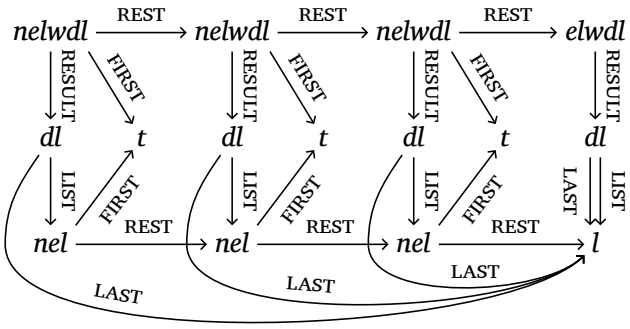


Figure 33: Illustration of *list-with-diff-list*. Type names are abbreviated. The new list starts at RESULT|LIST, and finishes with an open end at RESULT|LAST.

Application: Multiple Extraction 5.4.5

TODO! expand

SLASH list. Important to know if empty (not specified with a diff-list, although technically possible to use a constraint to check, creating a cycle if the list is empty).

Zamaraeva and Emerson (2020).

Application: Valence Changes 5.4.6

TODO! expand

Explain typical HPSG feature geometry for valence. Valence changing in the Grammar Matrix Curtis (2019)... Appending to beginning is easy, appending to end is difficult.

Introducing additional layer of wrapping would allow immediate use of list operations.

Alternatively, the *valence* type can be seen as a wrapper type, simultaneously wrapping multiple lists. For any operation for any of the valence lists, we can define an appropriate subtype of *valence*. Combinations of these operations (e.g. one operation for each list) can be defined as further subtypes (with no additional constraints). In terms of the number of types and features required, this is more verbose than introducing a wrapper for each valence list, but it would mean that the feature geometry is backwards-compatible with existing work.

Nondeterministic Computation 5.5

In §5.2, we explained how we can implement relational constraints with wrapper types. However, this only works for relational con-

straints which correspond to deterministic functions (the value on the mother is determined by the values of the daughters).

A nondeterministic relational constraint allows multiple possible values on the mother. While multiple possibilities can often be expressed through underspecification, this is not always the case. For example, given a list, we may want to remove exactly one item, but not specify which. For a list of length n, there are n possible results. With our encoding of lists, an underspecified list that subsumes all n of these lists would also subsume other lists too.

Since we have a Turing-complete formalism, we could in principle come up with a canonical encoding of the set of possible values, and define a deterministic function that maps to this encoding. However, such encodings would be cumbersome in practical applications.

On the other hand, the phrase structure formalism described in §5.1 does provide nondeterminism outside of the type system: the application of phrase structure rules is nondeterministic, since different possible rules can apply at the some point. The aim of this section is to exploit this fact, so that the grammar will generate one edge in the parse chart for each possible result of the nondeterministic computation.

This is illustrated in Fig. 35, which invokes a nondeterministic computation via the feature NDET (unlike a wrapper type, the computation is specified at the top level, rather than adjacent to the data type). The ambiguity inherent to the nondeterministic computation is expressed via unary rules, as illustrated in Fig. 36.

We will explain how to define nondeterministic computation types in §5.5.1, explain how the computation can be carried out via unary rules in §5.5.2, suggest best practices in §5.5.3, and then apply this to modelling flexible word order in §5.5.4.

5.5.1 Underspecified Computation

As in previous sections, we will use computation types to define steps in a computation. However, unlike the constraints we have seen that propagate recursively (along a list or Peano numeral), the constraints in this section will "stop".

For example, types for nondeterministically popping one item from a list are shown in Fig. 37. The result has two parts: the list with

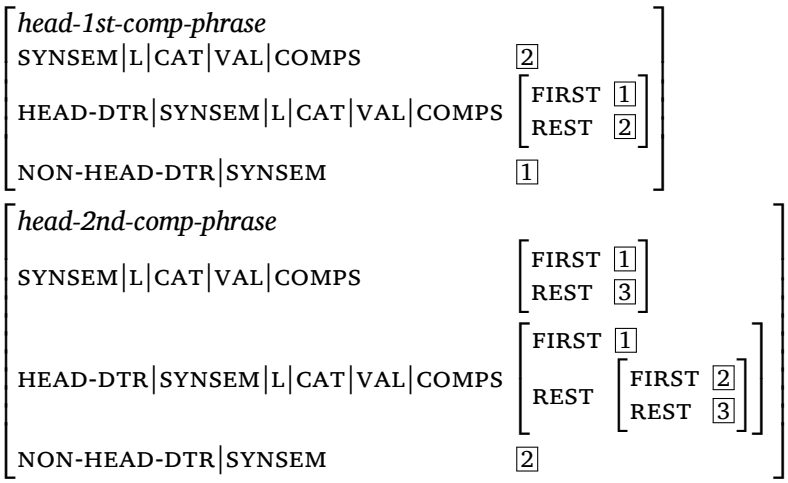


Figure 34: Deterministic head-comp rules, for the first or second element on the COMPS list, respectively.

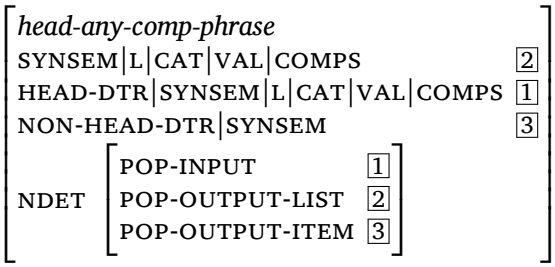


Figure 35: Nondeterministic head-comp rule, for any element on the COMPS list, generalising the rules in Fig. 34.

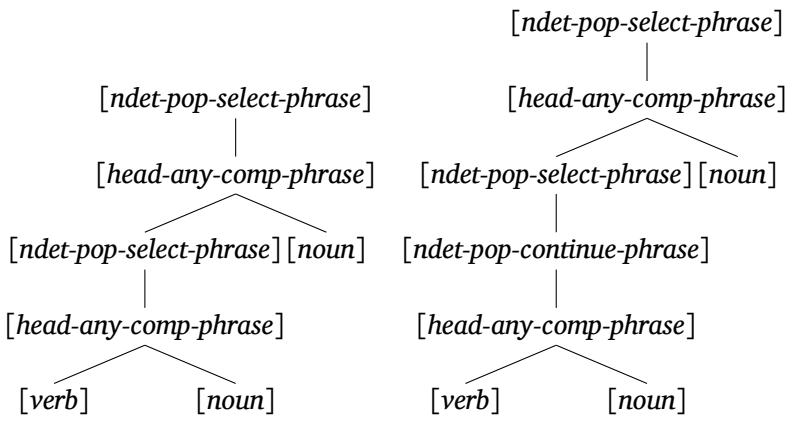


Figure 36: Phrase structure trees illustrating how word order ambiguity can be captured using nondeterministic computation types. The right-hand tree takes the second element on the COMPS list before the first.

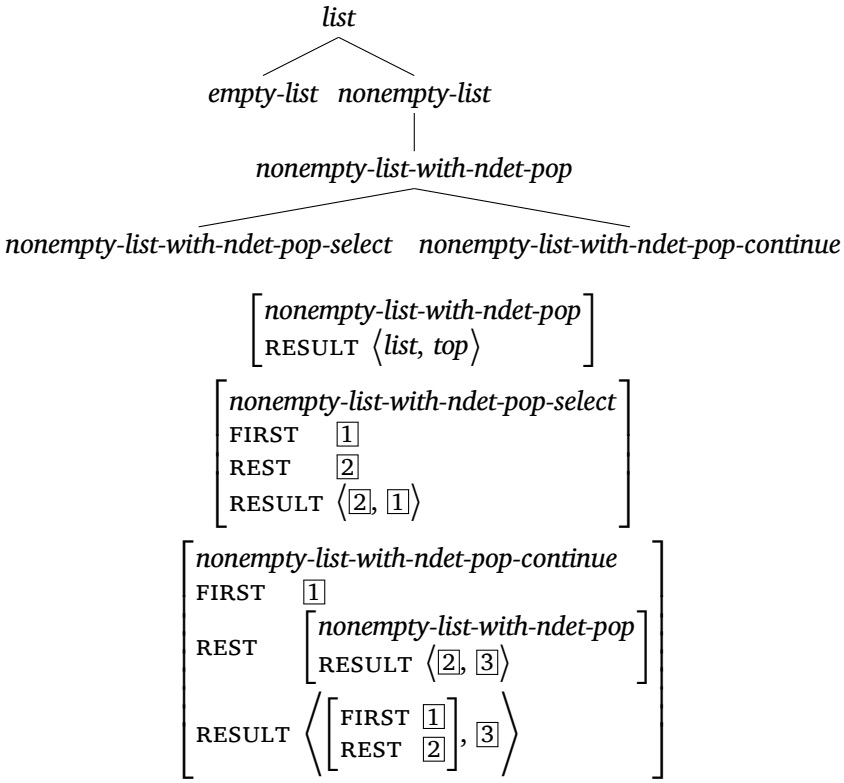


Figure 37: Computation types for nondeterministically popping an element from a list. The computation subtypes …*-select* and …*-continue* do not inherit from further data subtypes of *nonempty-list*, so they are not deterministically triggered by unifying *nonempty-list-with-ndet-pop* with a list.

one item popped, and the popped item. Unification fails on an empty list (there is no common subtype). For a nonempty list, we could either select the current element, or continue looking at the rest of the list. We therefore have two further subtypes. If we continue, we have the same two choices again, hence the recursive constraint on REST. However, unifying with the more general type (*nonempty-list-with-ndet-pop*) does not force either of the two more specific subtypes. In other words, recursion is underspecified.

Fig. 38 shows how a succession of unifications can give a specified result; in this case, the third element of 1 is popped, which would yield $\left[\textsc{result}\ \langle\langle \mathit{false}, \mathit{true}\rangle, \mathit{false}\rangle\right]$.

5.5.2 Unary Rules for Specifying Computation

In Fig. 38, the sequence of unifications is defined by a sequence of unary rules. Each step specifies a subtype on a particular feature path.

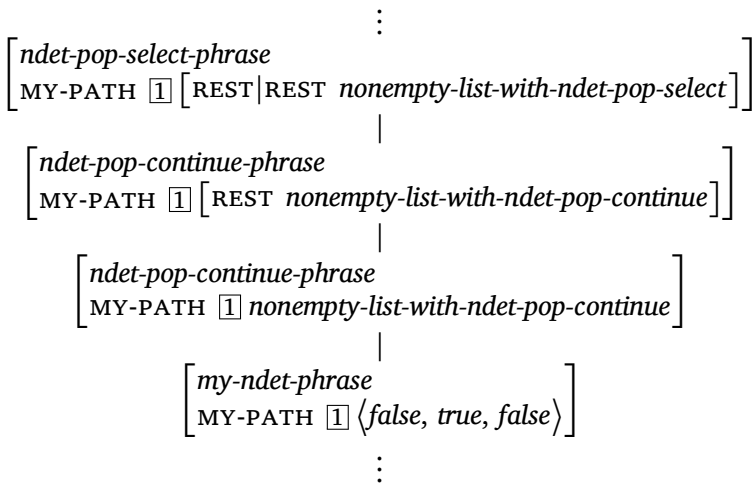


Figure 38: Sketch of how a sequence of unary rules can together specify a computation. The "input" at the bottom is unified with a succession of computation types, at different points along the list.

Effectively, the unary rules force the feature path to be sort-resolved, similarly to using junk slots (see §5.4.3).

However, because the specified subtypes are on different feature paths, some extra bookkeeping is necessary. In order to define unary rules that can specify subtypes in the appropriate places, we need to introduce a feature to track which feature path requires its type to be specified.

Because of this bookkeeping, nondeterministic computation requires a more complicated type system. However, once the type system has been defined, nondeterministic computation can be straightforwardly used in multiple places in a grammar. In the following section (§5.5.3), we suggest best practices for this bookkeeping. There is no formal status to these best practices (s in §5.2.1); rather, the aim is ease of use.

Suggested Best Practices 5.5.3

To encapsulate nondeterministic computation, we suggest using NDET as a top-level feature, with value *ndet*. The *ndet* type has subtypes *ndet-pending* and *ndet-satisfied*, indicating that a nondeterministic computation rule is expected or not expected, respectively. The *ndet-pending* type also introduces the POINTER feature, which is intended to be re-entrant with the feature path where a computation subtype needs

Figure 39: Feature geometry with bookkeeping for nondeterministic computation.

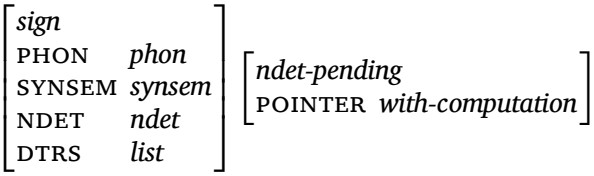
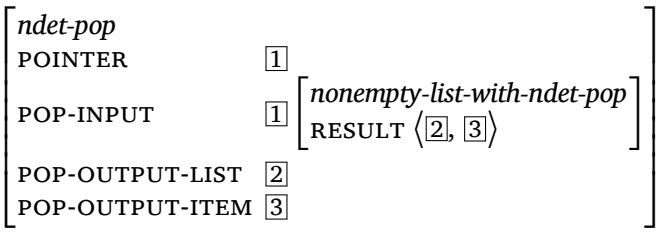


Figure 40: A subtype of *ndet-pending*, for popping some element of a list, to be invoked as illustrated in Fig. 35.



to be further specified.[27] This is shown in Fig. 39. Each subtype of *ndet-pending* defines a particular operation (such as *ndet-pop*, shown in Fig. 40), and should introduce one or more unique features (so that it can be invoked via the features, as illustrated in Fig. 35).

All lexical entries and all roots should specify NDET to be *ndet-satisfied*. All normal rules should specify this on all daughters, by inheriting from *basic-unary-phrase* or *basic-binary-phrase*, shown in Fig. 41.

Normal rules that do not invoke nondeterministic computation should specify NDET to be *ndet-satisfied*, by inheriting from *deterministic-phrase*, shown in Fig. 41. Normal rules can invoke a nondeterministic operation by specifying features of NDET with appropriate re-entrancies (as illustrated in Fig. 35).

Each subtype of *ndet-pending* should be associated with two or more nondeterministic computation rules. Each such rule should inherit from *ndet-computation-phrase*, shown in Fig. 42, and specify its daughter's NDET|POINTER with a subtype. Depending on whether further nondeterministic decisions are necessary, each rule should either specify NDET to be *ndet-satisfied*, or specify a re-entrancy for NDET|POINTER.

[27] This approach only allows one nondeterministic operation per phrase. If more operations are needed, we could have a list of pointers, but this could become difficult to maintain, particularly if there is multiple inheritance of nondeterministic rules. We do not anticipate such a need in practical grammars.

$$\begin{bmatrix} \textit{deterministic-phrase} \\ \textsc{ndet}\ \textit{ndet-satisfied} \end{bmatrix} \begin{bmatrix} \textit{basic-unary-phrase} \\ \textsc{dtrs}\ \left\langle \begin{bmatrix} \textsc{ndet}\ \textit{ndet-satisfied} \end{bmatrix} \right\rangle \end{bmatrix}$$
$$\begin{bmatrix} \textit{basic-binary-phrase} \\ \textsc{dtrs}\ \left\langle \begin{bmatrix} \textsc{ndet}\ \textit{ndet-satisfied} \end{bmatrix}, \begin{bmatrix} \textsc{ndet}\ \textit{ndet-satisfied} \end{bmatrix} \right\rangle \end{bmatrix}$$

Figure 41: Supertypes specifying that *ndet* unary rules are not needed, for a rule and for its daughters.

$$\begin{bmatrix} \textit{ndet-computation-phrase} \\ \textsc{phon}\ \boxed{1} \\ \textsc{synsem}\ \boxed{2} \\ \textsc{dtrs}\ \left\langle \begin{bmatrix} \textsc{phon}\ \boxed{1} \\ \textsc{synsem}\ \boxed{2} \\ \textsc{ndet}\ \textit{ndet-pending} \end{bmatrix} \right\rangle \end{bmatrix}$$
$$\begin{bmatrix} \textit{ndet-pop-select-phrase} \\ \textsc{ndet}\ \textit{ndet-satisfied} \\ \textsc{dtrs}\ \left\langle \begin{bmatrix} \textsc{ndet|pointer}\ \textit{nonempty-list-with-ndet-pop-select} \end{bmatrix} \right\rangle \end{bmatrix}$$
$$\begin{bmatrix} \textit{ndet-pop-continue-phrase} \\ \textsc{ndet|pointer}\ \boxed{1} \\ \textsc{dtrs}\ \left\langle \begin{bmatrix} \textsc{ndet|pointer}\ \begin{bmatrix} \textit{nonempty-list-with-ndet-pop-continue} \\ \textsc{rest}\ \boxed{1} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

Figure 42: Supertype for *ndet* unary rules that continue a nondeterministic computation, and two subtypes which together implement the nondeterministic pop operation. Their use is illustrated in Fig. 36.

Application: Flexible Word Order 5.5.4

TODO! expand

Bender (2008a,b, 2010). COMPS list, have to take off in specific order. Can use multiple head-comp rules, as in Fig. 34. This works as long as lists are limited in size, so that we can define enough rules to cover the longest possible list. If lists can be dynamically constructed (with no upper bound), then a different mechanism is necessary.

Instead, nondeterministic pop operation, where the head-comp rule can combine with any item on the COMPS list, as we saw in Fig. 35. Single rule that can work with arbitrarily long lists.

For example, with two elements on the COMPS list, there are two parses, as shown in Fig. 36.

6 CONCLUSION

We have shown how the formalism presented by Copestake (2000) allows functions to be encoded as subtypes. We have proven that one-feature type systems are finite-state, and two-feature type systems are Turing-complete. We have developed the notions of "computation type" and "wrapper type", and given best practices for working with them in large-scale grammars, in a way that mimics relational constraints. Finally, we have given several examples of how wrapper types can be used to analyse syntactic phenomena.

REFERENCES

Gabriel Aguila-Multner and Berthold Crysmann (2018), Feature Resolution by Lists: The Case of French Coordination, in Annie Foret, Greg Kobele, and Sylvain Pogodalla, editors, *Proceedings of the 23rd International Conference on Formal Grammar*, number 10950 in Lecture Notes in Computer Science, pp. 1–15, Springer.

Hassan Aït-Kaci (1984), *A lattice-theoretic approach to computation based on a calculus of partially-ordered type structures (property inheritance, semantic nets, graph unification)*, Ph.D. thesis, University of Pennsylvania.

Emily M. Bender (2008a), Evaluating a Crosslinguistic Grammar Resource: A Case Study of Wambaya, in *Proceedings of ACL-08: HLT*, pp. 977–985, Association for Computational Linguistics, https://aclanthology.org/P08-1111.

Emily M. Bender (2008b), Radical Non-Configurationality without Shuffle Operators: An Analysis of Wambaya, in Stefan Müller, editor, *Proceedings of the 15th International Conference on Head-Driven Phrase Structure Grammar*, pp. 6–24, CSLI Publications, https://doi.org/10.21248/hpsg.2008.1.

Emily M. Bender (2010), Reweaving a grammar for Wambaya, *Linguistic Issues in Language Technology*, 3(1).

Emily M. Bender, Scott Drellishak, Antske Fokkens, Michael Wayne Goodman, Daniel P. Mills, Laurie Poulson, and Safiyyah Saleem (2010), Grammar Prototyping and Testing with the LinGO Grammar Matrix Customization System, in *Proceedings of the ACL 2010 System Demonstrations*, pp. 1–6, Association for Computational Linguistics, https://aclanthology.org/P10-4001.

Emily M. Bender and Guy Emerson (2021), Computational linguistics and grammar engineering, in Stefan Müller, Anne Abeillé, Robert D. Borsley, and Jean-Pierre Koenig, editors, *Head-Driven Phrase Structure Grammar: The handbook*, number 9 in Empirically Oriented Theoretical Morphology and Syntax, chapter 25, pp. 1105–1153, https://zenodo.org/record/5599868/files/259-M%C3%BCllerEtAl-2021-25.pdf.

Gosse Bouma, Robert Malouf, and Ivan A Sag (2001), Satisfying constraints on extraction and adjunction, *Natural Language & Linguistic Theory*, 19(1):1–65, doi:10.1023/A:1006473306778.

Ulrich Callmeier (2000), PET – A platform for experimentation with efficient HPSG processing techniques, *Natural Language Engineering*, 6:99–108.

Bob Carpenter (1992), *The Logic of Typed Feature Structures, with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.

Ann Copestake (2000), Definitions of Typed Feature Structures, *Natural Language Engineering*, 6(1):109–112, reprinted in Stephan Oepen, Daniel Flickinger, Hans Uszkoreit and Jun'ichi Tsujii (2002), editors, *Collaborative Language Engineering: A Case Study in Efficient Grammar-based Processing*, pp. 227–230, CSLI Publications.

Ann Copestake (2002), *Implementing typed feature structure grammars*, volume 110 of *CSLI Lecture Notes*, CSLI Publications.

Berthold Crysmann and Woodley Packard (2012), Towards efficient HPSG generation for German, a non-configurational language, in Martin Kay and Christian Boitet, editors, *Proceedings of the 24th International Conference on Computational Linguistics*, pp. 695–710, Mumbai, http://www.aclweb.org/anthology/C12-1043.

Christian Curtis (2019), A Parametric Approach to Implemented Analyses: Valence-changing Morphology in the LinGO Grammar Matrix, in *Proceedings of the Second International Workshop on Resources and Tools for Derivational Morphology*, pp. 111–120, Charles University, Faculty of Mathematics and Physics, Institute of Formal and Applied Linguistics, Prague, Czechia, https://aclanthology.org/W19-8513.

Scott Drellishak and Emily M. Bender (2005), A coordination module for a crosslinguistic grammar resource, in Stefan Müller, editor, *Proceedings of the 12th International Conference on Head-driven Phrase Structure Grammar*, pp. 108–128, CSLI Publications, Stanford, CA, doi:10.21248/hpsg.2005.6.

Dan Flickinger, Carl Pollard, and Thomas Wasow (2021), The evolution of HPSG, in Stefan Müller, Anne Abeillé, Robert D. Borsley, and Jean-Pierre Koenig, editors, *Head-Driven Phrase Structure Grammar: The*

handbook, number 9 in Empirically Oriented Theoretical Morphology and Syntax, chapter 2, pp. 47–87, doi:10.5281/zenodo.5599820, `https://zenodo.org/record/5599820/files/259-M%C3%BCllerEtAl-2021-2.pdf`.

Antske FOKKENS, Yi ZHANG, and Emily M. BENDER (2011), Spring Cleaning and Grammar Compression: Two Techniques for Detection of Redundancy in HPSG Grammars, in *Proceedings of the 25th Pacific Asia Conference on Language, Information and Computation*, pp. 236–244, `https://aclanthology.org/Y11-1025`.

Nissim FRANCEZ and Shuly WINTNER (2011), *Unification grammars*, Cambridge University Press.

Ulrich GESKE and Hans-Joachim GOLTZ (2007), A guide for manual construction of difference-list procedures, in *Applications of Declarative Programming and Knowledge Management*, pp. 1–20, Springer.

Thilo GÖTZ and Walt Detmar MEURERS (1996), The importance of being lazy – Using lazy evaluation to process queries to HPSG grammars, in *Actes de la conférence Traitement Automatique de la Langue Naturelle (TALN)*.

John E. HOPCROFT and Jeffrey D. ULLMAN (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley.

Dakotah Jay LAMBERT (2022), *Unifying Classification Schemes for Languages and Processes With Attention to Locality and Relativizations Thereof*, Ph.D. thesis, Stony Brook University.

Sylvain LOMBARDY and Jacques SAKAROVITCH (2006), Sequential?, *Theoretical Computer Science*, 356(1–2):224–244.

M. LOTHAIRE (2005), *Applied combinatorics on words*, volume 105 of *Encyclopedia of Mathematics and its Applications*, Cambridge University Press.

Per MARTIN-LÖF (1984), *Intuitionistic type theory*, number 1 in Studies in Proof Theory, Bibliopolis.

Nurit MELNIK (2007), From "Hand-Written" to Computationally Implemented HPSG Theories, *Research on Language and Computation*, 5(2):199–236, `http://dx.doi.org/10.1007/s11168-007-9028-0`.

Walt Detmar MEURERS, Kordula DE KUTHY, and Vanessa METCALF (2003), Modularity of grammatical constraints in HPSG-based grammar implementations, in *Proceedings of the ESSLLI 2003 Workshop on Ideas and Strategies for Multilingual Grammar Development*, pp. 83–90.

Walt Detmar MEURERS, Gerald PENN, and Frank RICHTER (2002), A Web-based Instructional Platform for Constraint-Based Grammar Formalisms and Parsing, in Dragomir RADEV and Chris BREW, editors, *Proceedings of the Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, pp. 18–25, The Association for Computational Linguistics, doi:10.3115/1118108.1118111, `https://aclanthology.org/W02-0103/`.

Stefan MÜLLER (2015), HPSG – A Synopsis, in Artemis ALEXIADOU and Tibor KISS, editors, *Syntax – Theory and Analysis: An International Handbook*, volume 2, pp. 937–973, De Gruyter Mouton, doi:10.1515/9783110363708-004.

Gerald PENN (2004), Balancing Clarity and Efficiency in Typed Feature Logic Through Delaying, in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pp. 239–246, doi:10.3115/1218955.1218986, https://aclanthology.org/P04-1031.

Carl J. POLLARD and M. Andrew MOSHIER (1990), Unifying partial descriptions of sets, in Philip P. HANSON, editor, *Information, Language, and Cognition*, number 1 in Vancouver Studies in Cognitive Science, pp. 285–322, University of British Columbia Press.

Carl J. POLLARD and Ivan A SAG (1994), *Head-Driven Phrase Structure Grammar*, University of Chicago Press.

James PUSTEJOVSKY (1995), *The Generative Lexicon*, MIT Press.

Marcel-Paul SCHÜTZENBERGER (1977), Sur une variante des fonctions séquentielles, *Theoretical Computer Science*, 4(1):47–57.

Glenn C. SLAYDEN (2012), *Array TFS storage for unification grammars*, Master's thesis, University of Washington.

Gabriele VANONI (2022), *On Reasonable Space and Time Cost Models for the λ-Calculus*, Ph.D. thesis, University of Bologna.

Olga ZAMARAEVA, Chris CURTIS, Guy EMERSON, Antske FOKKENS, Michael Wayne GOODMAN, Kristen HOWELL, TJ TRIMBLE, and Emily M. BENDER (2022), 20 years of the Grammar Matrix: cross-linguistic hypothesis testing of increasingly complex interactions, *Journal of Language Modelling Vol*, 10(1):49–137.

Olga ZAMARAEVA and Guy EMERSON (2020), Multiple Question Fronting without Relational Constraints: An Analysis of Russian as a Basis for Cross-Linguistic Modeling, in *Proceedings of the 27th International Conference on Head-Driven Phrase Structure Grammar*, pp. 157–177, CSLI Publications.