# Program Search as a Path to Artificial General Intelligence

Łukasz Kaiser

Mathematische Grundlagen der Informatik, RWTH Aachen
D-52056 Aachen, Germany
lukaszkaiser@gmail.com

**Summary.** It is difficult to develop an adequate mathematical definition of intelligence. Therefore we consider the general problem of searching for programs with specified properties and we argue, using the Church-Turing thesis, that it covers the informal meaning of intelligence. The program search algorithm can also be used to optimise its own structure and learn in this way. Thus, developing a practical program search algorithm is a way to create AI.

To construct a working program search algorithm we show a model of programs and logic in which specifications and proofs of program properties can be understood in a natural way. We combine it with an extensive parser and show how efficient machine code can be generated for programs in this model. In this way we construct a system which communicates in precise natural language and where programming and reasoning can be effectively automated.

## 1 Intelligence and the Search for Programs

Intelligence is usually observed when knowledge is used in a smart and creative way to solve a problem. Still, it seems that the core of intelligence is neither the knowledge nor the specific method to use it, but the general way to learn from previous experience. This is not limited to adopting new knowledge, but also includes learning new ways to use what we know, extending it by reasoning, and even improving learning methods to learn more efficiently. Developing new ways to solve problems is a better indication of intelligence than solving separate tasks, as it is a creative work, where we do not have a precise description of what to do and are expected to find the right method knowing only what goals we want to achieve.

We will represent the informal notion of learning new ways to solve problems as the search for programs that fulfil some properties and we will design a system to make it practical. To explain why we choose this representation we have to analyse how methods of solving problems in general can be modelled by abstract notions and how problems can be specified. We use the general representation that dates back to the birth of AI and computer science with the works of Gödel, Turing, and Church.

We claim that the informal notion of a method for solving certain tasks can be expressed in mathematical terms as a Turing machine. To justify this

we use the Church-Turing thesis, the assumption that everything that is computable, any complex behaviour of a system, can be computed or modelled using only a small set of simple abstract operations. We can take different sets of such operations, use either Turing machines or lambda calculus, recursive functions or any other programming language. Still, these all have the same computational power and over fifty years after stating this thesis we did not manage to find any physical system, neither classical nor quantum, that would be able to compute more than a simple Turing machine. Note a straightforward consequence of the Church-Turing thesis: as far as we assume that humans are normal, although very complex physical objects, the procedure that operates in our brains can also be implemented on Turing machines and therefore also on usual computers with enough memory, when these get fast enough.

The thesis of Church and Turing justifies that any informally understood *method for solving a problem* can be defined as an algorithm, a Turing machine that takes the instance of the problem as input and returns the solution.

Of course, to be considered a viable solution for the given problem the method (now – the Turing machine) has to fulfill certain requirements that depend on the problem. For example, if we want to find a way to sort cards, there might be many better or worse ways to do this, machines that take the cards and return them mixed, but any *solution* must return the cards in the right order. We will use the natural (first order) logic with the language appropriate for describing Turing machines to specify such requirements.

Please note that in this logic we are not only able to specify what a good solution is; we can also define an ordering, defining when one solution is better than another. We can say, for example, that solution A is better than solution B if it takes less time to sort through the cards, and this can be expressed using the definition of the number of steps in a run of a Turing machine. We also have to take into account that often the goals to achieve or the conditions of work will not be directly specified, but can refer to knowledge about similar events in the past. This can also be included in our requirement specification if we encode the past knowledge inside the formula. Since we assume the Church-Turing thesis, we can also take it for granted that a Turing machine can verify the correctness of a solution, and then all possible problems that an intelligent agent will ever be required to solve can be specified in first order logic, or even a limited variant of it.

We have modeled problem solving as searching for Turing machines with specified properties. Determining if such a machine exists is of course undecidable and the problem is intractable in general, but we can make some additional assumptions. First, we can assume that we do not only want the machine, but also a proof that it satisfies the formula and that such a machine with a proof exists. This is a realistic assumption in the context of artificial intelligence, since the agent normally wants to solve a problem that is solvable, and when the solution is found then it should be clear that it is correct. When no solution can be found or the agent knows nothing about whether

it is correct or not, not even in the probabilistic sense, then it has to resort anyway to other methods that we do not investigate here, like asking another agents for help, or trying to solve the problem again later. Therefore, we will not consider the cases when the problem is not solvable or it can not be proved that the solution is correct, since in such cases the AI agent has to determine when to stop searching for the solution using external knowledge and taking other factors into account. Instead, we will concentrate on making a model of programs and a program search algorithm that preserves generality, yet is simple and efficient enough to be used in practice for specific classes of problems.

As we mentioned discussing intelligence, we do not only want a procedure to solve certain tasks, but we want the agent to learn. Learning, in this case, amounts to improving the procedure, so that after a number of problem instances have been solved it will solve other similar instances more efficiently. We will present a self-improving algorithm that searches for Turing machines with specified properties. Moreover, we will show an innovative system that binds programming and problem solving with natural language processing.

**Outline.** In the next section we will look for a general procedure that, when given a logic formula, looks for a Turing machine fulfilling it, and that optimizes itself with each successful run. We will present such a theoretical method based on the program and proof enumeration technique, which was already used by Gödel [4] and Turing. The resulting procedure has the nice property of self-improvement, similarly to how we improve our learning skills, and it is very general, so after some time it will become as good as any other such procedure with respect to any appropriate measure of efficiency. We will also show how it can be used by an AI agent in an unknown environment to learn to take successful actions.

The problem we face with such a theoretical solution is that it would not be usable in practice if implemented in a direct way. The time required for it to improve to a level of efficiency that would give any tangible results would be enormous. Therefore, in subsequent sections we will present a model of computation and program logic that combines functional programming with reasoning using games. This model is powerful enough to express algorithms and proofs on the same level of abstraction as we think of them, and at the same time compile programs to binary code. Thus, when running the program search procedure in this model, we can expect the implementation to execute efficiently and, even when it does not find the results automatically, we can still understand the steps it takes and guide it to the correct solution.

In Sect. 3 we will present the model and additionally give a method to parse compound expressions that fits in the model. Such parsing improves the presentation of programs and proofs, and can be extended to handle basic natural language processing. We will also use examples to show the compilation of programs from this model to efficient code, going through the C language.

In Sect. 4 we will analyse how properties of programs described in the model can be proved formally at a high level of abstraction. We will show

how automatic proofs can be guided by the user or by different heuristics, and how sub-procedures for reasoning in less general cases can be included in the model without loss of generality.

Please note that the theoretical results we present are well known and we do not discuss them very precisely. The model of computation, the method to parse expressions, and the logic presented later are also based on well known ideas but their combination is innovative. Therefore, we give more details about it and describe how to create a system that allows to write in natural language programs about which we can reason semi-automatically in formal logic, and which can be compiled to efficient machine code.

## 2 Theoretical Results

In this section we give an overview of the theoretical results that concern searching for programs with specified properties, and using program search in the standard AI model. We take Turing machines as our model of computation but any other Turing-complete model could be used here. Also, we do not give the results in full detail, as most of them are already standard knowledge in computer science, and we just want to put them in the context of AGI or extend them, and in such cases we give references to papers where these extensions are thoroughly discussed.

We start our theory by setting a description of programs and choosing a computable set of axioms from which we will deduce program properties. Later, we will present a model of programs that we consider simple and more practical, but let us now consider the Turing machines defined in set theory together with the axioms of set theory as formalized by Zermelo and Fränkel, which is a widely used axiomatization.

The *program search problem* can be stated as follows: given a formula $\varphi(x_1, \ldots, x_n)$ in first order logic on the structure defined above with free variables $x_1, \ldots, x_k$ denoting Turing machines, find a proof of $\varphi(m_1, \ldots, m_k)$ for some Turing machines $m_1, \ldots, m_k$.

Let us now state an important positive fact which is a straightforward consequence of the enumerability of Turing machines and proofs.

**Fact 1.** *There exists an algorithm that computes the solution to the program search problem if any solution exists, so given $\varphi(x_1, \ldots, x_k)$ it computes $m_1, \ldots, m_k$ and the proof of $\varphi(m_1, \ldots, m_k)$, assuming that for some machines such a proof exists.*

*Proof.* Since Turing machines, programs, and proofs are enumerable and it can be determined algorithmically whether a sequence of formulas forms a proof of a given claim, we can use the following algorithm to prove this fact:

(1) Set `length` to 1.

(2) Enumerate all $k$-tuples $m_1, \ldots, m_k$ of Turing machines shorter that `length` and all proofs shorter than `length` and check if there is any proof among these that proves $\varphi(m_1, \ldots, m_k)$.
(3) If the correct machines and proof were found, return them, else increase `length` by one and return to point (2).

Of course, this algorithm will find a solution, even the shortest one, if it exists. Otherwise, the algorithm will never stop. We will denote this algorithm by $\mathrm{PSP}_0$.

### 2.1 Program Search in the Standard AI Model

We will now consider the often used AI model where the agent interacts with the environment. The agent is modeled to have sensors from which it collects input, and effectors which it uses to execute actions. Additionally, at any moment the agent may get additional feedback that denotes its own happiness, or a quantified assessment it gets from a teacher agent. The agent's task is to maximize the total assessment it gets throughout its life.

To be able to construct well-acting agents we have to assume something about the environment, or, at least, something about its probabilistic behaviour. One sensible assumption is that the environment, or at least the probability distribution of events, is driven by some program (Turing machine). We want to create an agent that will behave in a worse way than the optimal agent, if one exists, only for some period of time, and that will later act optimally.

Let us sketch the possible construction of such an agent, which uses the program search to find rules in environment behaviour, and uses these rules as predictors, in order to find the best possible actions in the assumed environment. This is a very natural general way to act by first planning actions according to the expected future outcome, and then choosing the best ones. Let our agent store the following internal variables:

(i) a list of interwoven events and actions called `history`, initially empty;
(ii) a program `model` that models the environment, initially any short one;
(iii) a program `actor` that models the suspected optimal behaviour of the agent, initially any trivial program;
(iv) two numbers `max size` and `max time`, initially set to 1.

We consider a model of the environment $m_1$ to be better than $m_2$ if we can prove that there is an agent that achieves, using $m_1$, a better assessment than any agent can achieve using $m_2$. The agent will act according to the following algorithm when a new event is encountered.

(1) Append the event to `history`.
(2) Search for any program smaller than `max size` that generates `history` in less time than `max time`. Among such environment models, consider only the best ones as defined above, and update `model` to be one of the shortest of the best programs.

(3) Search for a proof, shorter than `max size`, that shows that some program, smaller than `max size` and halting on every input, can achieve a better assessment in environment `model` than the program `actor`. In that case update `actor` to be one of the shortest of such programs.
(4) Increase `max time` and `max size` by one.
(5) Calculate the response of `actor` to the input event, append the response to `history`, and output it.

Since in the construction we search through all possible programs, we can state the following simple fact.

**Fact 2.** *If a Turing machine can describe the behaviour of the environment and there is a provably optimal agent for this environment, then the presented agent gets assessment smaller than the optimal one only for some period of time, and behaves optimally afterwards.*

*Proof.* Indeed, if the environment is a program, then after some running time it will generate output that distinguishes it from any shorter program. Please note that before the model is clear, the agent will assume an optimistic one and undertake actions according to this assumption. Then, after analysing this output in step (2), the variable `model` will be set to the correct environment program. When this variable is set correctly the agent will search in step (3) for the optimal agent for the detected environment. Since we assumed that there is a provably optimal agent, this agent and the proof of its optimality have some length. When `max size` exceeds this length, the variable `actor` will be set to the optimal program. Therefore, the agent will start to behave optimally after detecting the correct environment and the necessary proof.

The construction of the AIXI agent, based on similar ideas, but extended and also specified in probabilistic context, was presented in detail and with full proofs of optimality by Hutter [7, 8], and the underlying theory is described thoroughly in [9]. The method to define different things as shortest possible programs was developed by Levin [13] in the framework of Kolmogorov Complexity theory [12, 21], and Li and Vitanyi give an excellent overview of these and similar methods in [14].

### 2.2 Self-improving Program Search

We saw that the program search problem can be useful for the construction of an AI agent, but we still do not know how to search for programs efficiently. We do not intend to search for any program in particular, but to learn efficient procedures to search for programs of interest. We will show how we can define what programs are interesting depending on the history of previous search tasks, and we will show how in such a case a procedure for program search can improve itself.

Let us therefore specify an algorithm that receives solvable instances of the program search problem, solves them, and improves its performance on such

and similar instances. To construct this procedure we need to define how to decide whether one program search algorithm is *more efficient* than another with respect to the history of observed instances of the problem, but we will postpone the discussion of such definitions until the next section. Also, the presented algorithm runs several processes simultaneously, but it is clear that such parallelism can be simulated on Turing machines as well as on single processor computers.

First, the algorithm initializes variable $P$ to $\mathrm{PSP}_0$, the program search algorithm presented before, and $P$ will be used both to solve received problem instances and for self-improvement. It also initializes `history` to an empty sequence. It then divides available resources into two parts and runs two processes simultaneously. Whenever a new instance of a program search problem is received, it is appended to `history`. The algorithm works with respect to the efficiency measure $\mu$ that in every moment depends on the `history` known at that moment.

When the **main** process receives the problem instance, it uses $P$ to solve it, and returns the solution.

The **improvement** process works as follows:

(1) Append the formula that describes the problem of creating a program search algorithm more efficient than $P$ with respect to $\mu$ to `history`.
(2) Use $P$ to find a more efficient program search algorithm as defined by the above formula.
(3) Update $P$ to a new, more efficient version.
(4) Repeat, starting from (1) with new $P$ and perhaps an extended `history`.

It can be seen that this algorithm not only solves the program search problem, but also uses its program search capacity to optimize itself. Therefore, even if $\mathrm{PSP}_0$ is not an efficient solution, the presented procedure will automatically find a better one, thanks to the improvement component. We assumed that the efficiency relation depends on the history. If we do not want this algorithm to fall in cycles thinking that some program search algorithm $P_1$ is better than $P_2$ and later, when history changes, deciding the other way, we have to assume that the definition of efficiency will be monotonic in some way. If we are not able to make such assumptions, it could be useful to separate the history of instances received from outside from the self-improvement instances, and use two separate program search algorithms, one for solving the problems and the second to improve program search. The following fact can be stated with the assumption that the definition of efficiency is appropriate, but extensions to more complex situations are also possible.

**Fact 3.** *Let a program search algorithm $Q$ (our goal, the efficient algorithm) be given and assume that the efficiency relation is such that there is only a bounded number of algorithms that are provably more efficient than* $\mathrm{PSP}_0$ *and less efficient than $Q$, with respect to any possible histories. Then, for any sequence of received instances, the presented algorithm will after some number*

*of steps substitute $Q$ for its internal variable $P$ and therefore become at least as efficient as $Q$.*

This way, if we find some reasonable definition of efficiency, then we can just start this algorithm and wait until it finds a good solution to the program search problem, which can then be used as an artificial general intelligence. The only practical issue is that if we start with $\mathrm{PSP}_0$ then even with the best computers we would have to wait very long. Similar learning algorithms and program searches have been analysed with the tools of Kolmogorov Complexity theory, see [14, 8] for more information on this topic. Schmidhuber gives detailed discussion of a recently developed optimally self-improving machine, called the Gödel Machine, in [19]. Such methods can also be relevant for physics as is discussed in [20].

### 2.3 Discussion of Efficiency Definitions

Let us now address the definition of the efficiency of algorithms which solve the program search problem. We will try to compare such algorithms with respect to a history of instances of the problem they solve.

The usual definitions of complexity, even in the asymptotic sense, can not be used in this case, as many instances are not solvable at all.

Let us again look at the problem from an informal and intuitive perspective. After gaining experience on a class of instances in the past, we will normally say that an algorithm is efficient if it solves the instances from this class and other *similar* instances fast. The remaining problem is to define which instances are similar. It seems reasonable to say that two instances are similar if one can be transformed into the other using a few simple transformations, for example by changing some parameters or shifting them in some way.

Assume that a set of simple transformations is given. Then we can define the *level of similarity* between two instances as the number of transformations that have to be applied to get from one instance to the other. For practical reasons we could also assume that if this number is greater than some constant, then the instances are not similar at all.

Using this, we can say that one program search algorithm is more efficient than another with respect to a history if it is faster on all instances in the history and on all similar instances. We could also use an alternative definition and say that the *weight* of an algorithm $A$ with respect to history $H$ is

$$w(A,H) = \Sigma_{\{i \text{ similar to some } j \in H\}} \mathrm{time}(A,i) \cdot 2^{\mathrm{similarity}(i,H)},$$

where $\mathrm{similarity}(i,H)$ denotes the smallest level of similarity between $i$ and any instance from $H$, and $\mathrm{time}(A,i)$ denotes the time it takes A to solve i. We assume that the sum is taken only over solvable instances $i$.

These two definitions seem reasonable and the first one satisfies the requirements presented in Fact 3, since it is monotonic with respect to history.

But, in practice, the second definition might be more useful, since it seems practical to decrease the efficiency of the algorithm in a few cases if it can lead to large improvements in other cases. It could also be practical to use some other weight for the definition of efficiency, for example including a heuristic that might make the efficiency a little worse in most cases, but improve it dramatically for some narrow class of cases.

Similar problems in the context of program search are considered in more detail by Schmidhuber in [18, 19], where more examples are presented. Still, it seems that the efficiency functions will have to be fine-tuned experimentally when such procedures start to be used in practice.

## 3 Convenient Model of Computation

We showed how to construct a learning program search procedure, but if we tried to implement it directly using $\mathrm{PSP}_0$, then it would not be practical. Therefore, our goal now is to present a more usable solution. The model we present with its theory is described in detail in the documents in [10], where the reader can also find an implementation of the discussed algorithms. Since this is still work in progress, and many details are actively being polished, the web site should be consulted for corrections and the most recent version. Many of these definitions and methods are already standard in functional programming and term rewriting [2].

Let us repeat our motivation: We need a model of computation which will allow us to easily write programs and, at the same time, reason about them. To construct such a model, we will concentrate only on two basic operations used in programming, namely the possibility to define and apply functions and the possibility to create compound data types. Therefore, in our model we will operate on objects that represent some data, e.g. $1, 2, [\mathrm{T}, \mathrm{F}]$, and on functions like $+, \cdot, \mathit{and}$. We are allowed to compose functions with data and write *terms* in this way, for example $1 + 2$, $\mathrm{T}\ \mathit{and}\ \mathrm{F}$ or $(1 + 2) \cdot (3 + 4)$.

To define functions in this model, we write rules telling how one term should change to another, e.g. $\mathrm{T}\ \mathit{and}\ \mathrm{F} \to \mathrm{F}$. In such rules we can use variables, for example, we can write $x + 0 \to x$. Note that not all terms have any meaning, for example $1 + \mathrm{T}$ does not mean anything. To avoid such terms we will introduce *types*, such that, for example 1 will have type *int* and $+$ will have type $\mathit{int}, \mathit{int} \to \mathit{int}$ so we will not be allowed to apply it to the boolean value T.

The model we present is known as term rewriting with polymorphic types. We will first give the basic definitions in detail, in order to show that formal reasoning about these objects is indeed feasible and to avoid confusion later, when we give examples less formally. We will also show how to parse terms from expressions in semi-natural language and how to generate efficient machine code for programs is this model. Thus, we will construct a computer

system where natural language input can be used for programming and reasoning without loss of efficiency of the created programs.

To define the model, we need the following classes, where *arity* is always a function that assigns a natural number to each element of the considered set:

(i) the infinite enumerable set of *type variables*, denote $\alpha, \beta, \gamma$;
(ii) the finite set $\Gamma$ of *type names* with arity, denoted $T, R, S$;
(iii) the infinite enumerable set $V$ of *term variables* with arity, denoted $x, y, z$;
(iv) the finite set $\Theta$ of *constructor names* with arity, denoted $A, B, C$;
(v) the finite set $\Sigma$ of *function names* with arity, denoted $f, g, h$.

**Types.** We start with formal type definitions. These might be difficult to understand at first, but the examples we give should be enough for an intuitive understanding. The set of types is defined inductively as the smallest set $\mathcal{G}$ such that:

(1) each type variable $\alpha \in \mathcal{G}$;
(2) if $T \in \Gamma$ with arity $n$ and $R_1, \ldots, R_n \in \mathcal{G}$ then $T(R_1, \ldots, R_n) \in \mathcal{G}$;
(3) for any number $n$ and types $T_1, \ldots, T_n \in \mathcal{G}$ and result type $R \in \mathcal{G}$ the *functional type* $(T_1, \ldots, T_n \to R) \in \mathcal{G}$.

We allow functional types for $n = 0$ to maintain consistent notation, but we consider the types $R$ and $\emptyset \to R$ to be identical, and we will not distinguish them.

Let us for example define the types of boolean values, pairs, and lists. We will set:

$$\Gamma = \{\text{booleans}, \text{lists}, \text{pairs}\},$$

where booleans has arity 0, lists arity 1 and pairs arity 2. Then, the example type $E$ of pairs consisting of a boolean value and a list of any other type can be represented as:

$$E = \text{pairs}(\text{booleans}, \text{lists}(\alpha)) \in \mathcal{G}.$$

The set $\text{TVar}(T)$ of type variables occurring in a type $T$ is also defined inductively by $\text{TVar}(\alpha) = \{\alpha\}$, $\text{TVar}(T(R_1, \ldots, R_n)) = \text{TVar}(R_1) \cup \cdots \cup \text{TVar}(R_n)$, and $\text{TVar}(T_1, \ldots, T_n \to R) = \text{TVar}(T_1) \cup \cdots \cup \text{TVar}(T_n) \cup \text{TVar}(R)$, so $\text{TVar}(E) = \{\alpha\}$.

The usual intuition behind types is to view them as labeled trees, therefore we introduce the notion of positions in types. The set $\Lambda$ of *positions* is the set of sequences of positive natural numbers. By $\lambda \in \Lambda$ we will denote the empty sequence or the top (root) position in the type.

For a given type $T$ and position $p$ we either say that $p$ does not exist in $T$, or define the type at position $p$ in $T$ (denoted by $T|_p$) in the following inductive way:

(1) $\lambda$ exists in each type and $T|_\lambda = T$;
(2) $p = (n, q)$ exists in $S = T(R_1, \ldots, R_m)$ if $m \geq n$ and $q$ exists in $R_n$ and in such case $S|_p = R_n|_q$;

(3) $p = (n, q)$ exists in $S = T_1, \ldots, T_m \to R$ if either $m \geq n$ and $q$ exists in $T_n$ and in such case $S|_p = T_n|_q$, or $m + 1 = n$ and $q$ exists in $R$ and then $S|_p = R_q$.

A position $p$ is *above* some position $q$ if there exists a sequence $r$ of numbers such that $q = (p, r)$. In this case we also say that $q$ is *below* $p$. The *height* of a position is its length, and the height of a type is the maximal height of a position existing in this type. In the example type $E$, one can see that position 3 does not exist in $E$, but $E|_{2,1} = \text{lists}(\alpha)|_1 = \alpha$ and so $E$ has height 2.

**Substitutions and unifiers.** Sometimes we want to change a part of a type, and then we say that we *substitute* type $S$ in type $T$ *at position* $p$. As a result we get the type $R = T[S]_p$, such that for all positions $q$ not below $p$ that exist in $T$, it holds that $R|_q = T|_q$ and $R|_p = S$. Less formally, $R$ is just $T$ with the subtree at position $p$ replaced by $S$. *Substituting* type $S$ in type $T$ *for a variable* $\alpha$ is defined as substituting $S$ in $T$ at all positions $p$ where $T|_p = \alpha$. A *type substitution*, usually denoted with letters $\sigma, \tau, \rho$, is a set of pairs, each consisting of a type variable and a type, and such pairs are denoted by $\alpha \leftarrow T$. For a substitution $\sigma = \{\alpha_1 \leftarrow T_1; \ldots; \alpha_n \leftarrow T_n\}$ we will denote the set of variables substituted for by $\text{TVar}(\sigma) = \{\alpha_1, \ldots, \alpha_n\}$ and we will say that by *applying* $\sigma$ to a type $T$ we obtain the type $R = T\sigma$, which is the result of substituting, for each $i$, the type $T_i$ in $T$ for the variable $\alpha_i$. In some algorithms it is necessary to ensure that the variables substituted for are disjoint with variables in the terms we substitute. As an example, let us apply $\{\alpha \leftarrow \text{booleans}\}$ to the type $E$ defined before and get

$$\text{pairs}(\text{booleans}, \text{lists}(\alpha))\{\alpha \leftarrow \text{booleans}\} = \text{pairs}(\text{booleans}, \text{lists}(\text{booleans})).$$

Sometimes we need to rename type variables in a type $T$; either all variables or only the variables from a given set $\mathcal{V}$. Let us set:

$$\sigma = \{\alpha \leftarrow \alpha\underbrace{'''\cdots'''}_{k} \ : \ \alpha \in \text{TVar}(T) \cap \mathcal{V}\},$$

where $k$ is first set to 1 and doubles each time we rename any type. Then we can define the renamed type $\overline{T}^{\mathcal{V}} = T\sigma$ and if we want to rename all type variables, we will just write $\overline{T}$ for $\overline{T}^{\text{TVar}(T)}$. As the names of substituted variables change with the number $k$ with each renaming, we can be sure that any two types $R$ and $S$ have disjoint variables after renaming, $\text{TVar}(\overline{R}) \cap \text{TVar}(\overline{S}) = \emptyset$.

We can apply a type substitution $\sigma$ to another type substitution $\rho = \{\alpha_1 \leftarrow T_1; \ldots; \alpha_n \leftarrow T_n\}$ and obtain the substitution:

$$\rho\sigma = \{\alpha_1 \leftarrow T_1\sigma; \ldots; \alpha_n \leftarrow T_n\sigma\}.$$

We will say that a type substitution $\sigma$ is *more general* than $\rho$ if there is another substitution $\tau$ for which $\sigma\tau \subseteq \rho$.

Let us now take a set of tuples of types

$$\{(T_1, R_1, \ldots, S_1), \ldots, (T_n, R_n, \ldots, S_n)\}.$$

Any substitution $\rho$ such that $T_i\rho = R_i\rho = \ldots = S_i\rho$ for each $i$ is called a *unifier* of this set, and it is a well known and important fact that if there is any unifier, then there exists the most general one, which we will denote by:

$$\mathrm{mgu}\{(T_1, R_1, \ldots, S_i), \ldots, (T_n, R_n, \ldots, S_n)\}.$$

The most general unifier can be computed in polynomial time if we can represent types in the form of acyclic graphs, and in exponential time if we restrict the representation to trees, where identical sub-trees can not be compressed.

For example, it is easy to see that there is no unifier for:

$$\{(\mathrm{pairs}(\mathrm{booleans}, \alpha), \mathrm{pairs}(\mathrm{lists}(\beta), \gamma)\},$$

but the pair of types $(\mathrm{pairs}(\alpha, \mathrm{booleans}), \mathrm{pairs}(\mathrm{lists}(\beta), \gamma)$ can be unified, and:

$$\mathrm{mgu}\{(\mathrm{pairs}(\alpha, \mathrm{booleans}), \mathrm{pairs}(\mathrm{lists}(\beta), \gamma)\} = \{\alpha \leftarrow \mathrm{lists}(\beta), \gamma \leftarrow \mathrm{booleans})\}.$$

When given a set of type substitutions $\{\sigma_1, \ldots, \sigma_n\}$, we will also use the most general unifier of these substitutions, $\tau = \mathrm{mgu}\{\sigma_1, \ldots, \sigma_n\}$, defined as the unifier of the set of tuples $(T_1^\alpha, \ldots, T_k^\alpha)$ of all such types that $\alpha \leftarrow T_i^\alpha \in \sigma_{l_i}$ for some $\sigma_{l_i}$, so all types substituted for the same variable in all substitutions $\sigma_i$ will be unified. Let us also denote, for each type variable $\alpha$, the unified type $T_1^\alpha\tau$ by $T^\alpha$ and let:

$$\mathrm{subst}\{\sigma_1, \ldots, \sigma_n\} = \{\alpha \leftarrow T^\alpha \ : \ \alpha \in \mathrm{TVar}(\sigma_1) \cup \ldots \cup \mathrm{TVar}(\sigma_n)\}.$$

**Typed terms.** We will now assume that each term variable $x \in V$, each constructor $C \in \Theta$, and each function $f \in \Sigma$ with arity $n$ has an associated functional type

$$\mathrm{type}(f)\ (\mathrm{type}(C),\ \mathrm{type}(x)) \ = \ T_1, \ldots, T_n \to R \ \in \mathcal{G}.$$

We will make an additional assumption that, for constructors, the type $R$ is neither a type variable nor a functional type, and has height at most one. Using this information about type we can define inductively the set of *well typed terms* $\mathcal{T}$, giving at the same time the definition of the type of a term, $\mathrm{type}(t) \in \mathcal{G}$, the set of variables of a term, $\mathrm{Var}(t) \subseteq V$, and the substitution $\rho(t)$ that reconstructs type variables in $\mathrm{Var}(t)$. To make the definition easier to follow, we will analyse the typing of the term $\mathrm{Pair}(y, y)$ with $\mathrm{type}(y) = \alpha$, and the constructor $\mathrm{Pair} \in \Gamma$ with type $\alpha, \beta \to \mathrm{pairs}(\alpha, \beta)$. We are using this slightly non-standard definition with reconstruction because it makes it easier to present the parsing algorithm later.

First, each variable $x \in V$, constructor $C \in \Theta$, and function symbol $f \in \Sigma$ belongs to $\mathcal{T}$ with the associated type, $\mathrm{Var}(C) = \mathrm{Var}(f) = \rho(C) = \rho(f) = \emptyset$, and

$$\mathrm{Var}(x) = \{x\},\ \rho(x) = \{\ \alpha \leftarrow \alpha \text{ for } \alpha \in \mathrm{TVar}(\mathrm{type}(x))\ \},$$

So, in our example, we have $\rho(y) = \{\alpha \leftarrow \alpha\}$.

Let a variable $x \in V$, constructor $C \in \Theta$, or function symbol $f \in \Sigma$ have arity $n > 0$. We will first rename the associated type $S$ and denote $\overline{S} = S_1, \ldots, S_n \rightarrow R$. At this point in our example, we rename the type of Pair to be $\alpha', \beta' \rightarrow \mathrm{pairs}(\alpha', \beta')$, thus expressing the fact that the type variable $\alpha$ is only accidentally the same in the type of $y$ and Pair.

Furthermore, let us take terms $t_1, \ldots, t_n \in \mathcal{T}$ with $\mathrm{type}(t_i) = R_i$ and rename all variables that are not reconstructed, so let

$$T_i = \overline{R_i}^{\mathrm{TVar}(R_i) \setminus \mathrm{TVar}(\rho(t_i))}.$$

In our example we do not rename anything, as we take $t_1 = t_2 = x$ and in $x$ all type variables are reconstructed. Then, $f(t_1, \ldots, t_n)$, $C(t_1, \ldots, t_n)$ or $x(t_1, \ldots, t_n)$ is well typed if there exists

$$\rho = \mathrm{mgu}\{\rho(t_1), \ldots, \rho(t_n)\} \text{ and } \sigma = \mathrm{mgu}\{(T_1\rho, S_1), \ldots, (T_n\rho, S_n)\},$$

and in such case, if $\tau = \mathrm{subst}\{\rho(t_1), \ldots, \rho(t_n)\}$ then

$$f(t_1, \ldots, t_n) \in \mathcal{T},\ \mathrm{type}(f(t_1, \ldots, t_n)) = R\sigma,$$

and $\mathrm{Var}(f(t_1, \ldots, t_n)) = \mathrm{Var}(t_1) \cup \cdots \cup \mathrm{Var}(t_n)$, $\rho(f(t_1, \ldots, t_n)) = \tau\sigma$, and likewise in the case of constructor $C$.

In our example the unifier $\rho$ of variable substitutions for $y$ is an empty substitution and $\sigma$ unifies both $\alpha'$ and $\beta'$ from the renamed type of Pair with $\alpha$. Then, substituting it in the pair type we get the result type $\mathrm{pairs}(\alpha, \alpha)$.

In the case of a variable, we have to extend the definitions, so we have $\mathrm{Var}(x(t_1, \ldots, t_n)) = \{x\} \cup \mathrm{Var}(t_1) \cup \cdots \cup \mathrm{Var}(t_n)$, and

$$\rho(x(t_1, \ldots, t_n)) = \tau\sigma \cup \{\alpha \leftarrow \alpha \text{ for } \alpha \in \mathrm{TVar}(R) \setminus \mathrm{TVar}(S_1) \cup \cdots \cup \mathrm{TVar}(S_n)\}.$$

We can also define positions in terms and term substitutions in an analogous way to the definitions for types, and we will say that a term $t$ is *ground* if $\mathrm{Var}(t) = \emptyset$, and that it is *linear* if no variable occurs in it at more than one position.

Let us for example take two constructors `T` and `F` with arity 0 and booleans as the assigned type. Let us also take the constructor `Pair` that we already know and two constructors for lists, `Nil` with arity 0 and type $\mathrm{lists}(\alpha)$, and `Cons` with arity 2 and type $\alpha, \mathrm{lists}(\alpha) \rightarrow \mathrm{lists}(\alpha)$.

In the examples we will use the symbol `:` to denote the type of a given term. We can now create terms with specific types, for instance:

```
Cons (T, Nil) : lists (booleans),
Pair (T, F) : pairs (booleans, booleans),
```

but we are not allowed to use terms that are not well typed, like

`Cons (F, T) or Cons (Pair (T, F), Cons (T, Nil)).`

In the first case, a term from booleans is used where a term from lists($\alpha$) is expected. In the second case, there is no correct type to instantiate the type variable $\alpha$ in `Cons` type definition, since there are both terms from booleans and pairs(booleans, booleans) in the list. Since we will be continuing this example, let us simplify our notation. We will denote `Cons (x, y)` by `x :: y` and `Pair (x, y)` by `(x, y)`, so the four terms presented above will be denoted:

`T :: Nil , (T, F) , F :: T and (T, F) :: (T :: Nil).`

To clarify the need for reconstructing substitutions, let us assume that we have three variables $x, y, z \in V$ with $\text{type}(x) = \text{lists}(\beta)$, $\text{type}(y) = \text{booleans}$, $\text{type}(z) = \gamma$. Now let us take the following example of two terms:

```
(x :: Nil, y :: Nil),
(x :: z :: Nil, y :: z :: Nil).
```

The first term is well typed, since `Nil` is a constructor and its type can unify with lists($\beta$) in one place and booleans in another as the type variable in type(Nil) will be renamed. The second term is not well typed since it is not possible to reconstruct the type for variable $z$, which can not have type booleans in one place and lists($\beta$) in another.

**Rewriting.** To define a function in a program that we want to execute, let us introduce the concept of a *rewrite rule*, a pair of terms $l$ and $r$, the left and right side of the rule, denoted by $l \to r$. In a rewrite rule $l \to r$ it must hold that $\text{Var}(l) \supseteq \text{Var}(r)$, $\text{type}(l) = \text{type}(r)$ (modulo renaming of type variables), and the symbol at the top position in $l$ must be a function name.

A rewrite rule $l \to r$ can be *applied* to a term $t$ at position $p$ if there exists a substitution $\sigma$ of variables in $l$ such that $t|_p = l\sigma$. The result of applying the rule is $t[r\sigma]_p$, the term $t$ rewritten at position $p$. Note that there is only one possible result of applying the rule to a term at a given position, and that the conditions guarantee that a ground term remains ground after applying the rule to it at any position, and in such a case it still has the same type after the rule is applied. The rule is ground if $r$ is ground and it is linear if $r$ is linear.

We will model programs by a system of terms and types defined above and a set of rewrite rules, where each subset of rules with the same function symbol at the top position on the left side is linearly ordered, and as you will see, the first rules in this order will be considered more important. Given a set $\mathcal{R}$ of rewrite rules, we say that a term $t$ rewrites to term $s$ in one step if there is a rule $l \to r \in \mathcal{R}$ that can be applied to $t$ at some position $p$ to give $s$, and two more conditions are fulfilled. First, no rule from $R$ can be applied to $t$ at a position below $p$, which is called *eager rewriting*. We make just one exception to this rule: the `if` function does not have to evaluate all branches. Second, no rule $l_1 \to r_1 \in \mathcal{R}$ with the same function name at the top position on the

left side, and before $l \to r$ in the linear order on such rules, can be applied to $t\tau$ at position $p$ for any substitution $\tau$ that could generate a conflict. We say that $\tau$ and $l_1 \to r_1$ generates a conflict with $l \to r$ on $t$ if $l_1 = t\tau\rho_1$, $l = t\tau\rho_2$ and $r_1\rho_1 \neq r\rho_2$. In this way we forbid the application of rules that are less important if any more important rule could be potentially applied and yield a different result. If $t$ contains function symbols then we treat them as variables, since the result of the function is unknown if it could not be rewritten.

The term $t$ rewrites to $s$ in $k$ steps if there is a term $u$ to which $t$ rewrites in $k-1$ steps and $u$ rewrites to $s$ in one step. We will also say that a term $t$ is in normal form if it can not be further rewritten. It follows from the linear order of rules with the same function symbol and the assumption of eager rewriting that if any term $t$ rewrites in any number of steps to a normal form, then $t$ does not rewrite to any other normal form.

We will now define the concatenation function from $\text{lists}(\alpha) \to \text{lists}(\alpha)$ that takes two lists and produces the concatenation of these lists, and to do this we need the function symbol $\text{concat} \in \Sigma$ and three variables $x, y, z$ with arity 0, $\text{type}(y) = \alpha$ and $\text{type}(x) = \text{type}(z) = \text{lists}(\alpha)$. The function can then be defined with the following two rewrite rules:

```
concat (Nil, x) -> x,
concat (x :: y, z) -> x :: concat (y, z).
```

To see how we execute the function let us concatenate T :: Nil with F :: Nil by rewriting the term, which is done in the following way:

$$\text{concat}(\text{T} :: \text{Nil}, \text{F} :: \text{Nil}) \to \text{T} :: \text{concat}(\text{Nil}, \text{F} :: \text{Nil}) \to \text{T} :: \text{F} :: \text{Nil}.$$

Term rewriting with types as presented above is used as the foundation for high level programming languages such as ML and Haskell, so the presented model is not only a precise mathematical entity that can be used for logical reasoning; it can be used to write programs that are easy to read and understand. Programs in other models of computation suitable for logical reasoning, like the Turing machines, are not directly readable. On the other hand, it is quite difficult to construct an elegant logical calculus for any imperative programming language used in practice and to reason about it.

**Term Rewriting I/O.** One problem with the presented model is the definition of input and output, since in term rewriting there are no side effects. Therefore, we will assume that the system of types, terms and rewriting rules that we are working with is a computer program and can respond to a set of commands that we will describe. There are commands that allow us to define new types, constructors, function symbols, and variables; commands that add new rewrite rules and rewrite terms. These will be discussed in the next section together with the extended notation that we will use. Now let us look at the additional commands that allow storage of strings or sequences of definitions in files, which can also be loaded. Moreover, there is the internal knowledge database, where terms of any type can be stored. Let `path` be a

string representing a path to a file or a virtual device, for example a printer or a display, `string` and `name` be strings, `type` be a type, and `term` be any term. The following commands provide input and output operations, with the last three used to define and manipulate storage space in the internal database:

```
Load string from [path].
Store [string] in [path].
Load definitions from [path].
Store system in [path].
Define data [name] in [type].
Load from [name].
Store [term] in [name].
```

When we define the storage space with the `define data` command we also set its type. This type must be more general than the type of anything we store using the `store` command, and it is the type of the term we get with the `load` command for this storage. To implement it without losing type correctness, we have to generate appropriate `load` and `store` commands for the internal database whenever the `define data` command is used.

In the presented setting, it is possible to load or store terms only after a complete sequence of rewriting steps; it is not possible to change the state of any variables during rewriting, which would complicate reasoning about programs. Since we normally rewrite terms between parsing, it is also necessary to add special handling of `load` commands when we construct system functions, because we have to prevent these commands from being in-lined before the actual call. We use a special tag `not_inline` in the function definition to prevent these functions from being evaluated before the correct time.

As terms are best suited for symbolic representation of data, the best way to create graphical programs in the system is to use vector graphics. We can connect the input and output of the term rewriting system with a HTML server and use the web standards like XForms and SVG. Then it is enough to generate terms with appropriate types corresponding to the specifications of the web standards and any browser can be used to run term rewriting programs with graphical interfaces. It is even better when this is combined with user interface ideas from [16] and, when the user can have all his work, both as text and graphical, on the desktop at once thanks to a zoomable interface [3].

### 3.1 Extended Program Notation

We showed a model of computation that suits our needs, but there is still a problem with the presentation of programs, as even for the short example we discussed it was necessary to define additional notation for list and pair constructors. Therefore, we will develop a more readable presentation that will be close to natural language. To enable this, let us assign syntax definitions to type names, constructors, function symbols and variables. We have to define a

*syntax element* as either a string or a type, and assume that the type of `types` is set. Sequences of syntax elements followed by a return type constitute syntax definitions.

For example, let us assign to the type name `lists` the following syntax definition: `'lists', 'of', types` with return type `types` and to the constructor `Cons` the following: `?a, ':' ':', lists (?a)` with return type `lists (?a)`. The meaning of the syntax definition in this constructor is the same as the notation we defined before.

Let us show an example of how a type name, constructor, a function symbol, and a variable are given with corresponding syntax definition. We will enclose the strings in syntax elements in stars `*` and use commands as in the following examples for definitions, where we assume that each type definition returns an element from `types`. In the definitions we can use the word `class` to denote types and the word `element` for constructors.

```
Define class *lists* *of* types.
Define element *Nil* in lists of ?a.
Define element ?a *:* *:* lists of ?a in lists of ?a.
Define variable *x* in lists of ?a.
Define function *concatenate* lists of ?a *with* lists of ?a
    into lists of ?a.
```

We will show how expressions can be parsed using syntax definitions, but note that the `define` command does not only add the defined type, constructor, variable or function name to the system and assigns the appropriate types and syntax definitions. Additionally, when the tag `is_functional` is specified, it adds a syntax definition that allows use of the constructors, functions or variables with arity bigger than 0 as functional values using just their names.

In this situation and when operating on function on meta level, which we discuss later, we need to have a single name corresponding to a syntax definition. These names are constructed automatically, in such a way that all the strings in the definition are kept and all types are changed to capitalized first letters of the type name preceeded with '; A is used for type variables. For example for the list constructor instead of `Cons` we will now use the name `'A_:_:'L_` and for list type definition the name `lists_of_'T_`. As the names have to be unambiguous, they always end with an underscore if only one syntax definition with corresponding name exists, and they end with a number, e.g. `lists_of_'T_1`, `lists_of_'T_2` if more definitions correspond to the same name.

Let us now show how we parse an input string and create a term from it. During parsing we use extended rewrite rules in the form $l_1, l_2, \ldots, l_n \rightarrow r$, which expect a sequence of terms and only if such a sequence is encountered, rewrite it to the resulting term $r$. Syntax definitions are easily encoded as such extended rewrite rules when each string is encoded as a term of type strings and each type $T$ is represented by a variable $x_T$ with $\text{type}(x_T) = T$. Let us

look, for example, at the extended rewrite rules for `lists` type definition and for the "`::`" constructor:

```
‘lists’ : strings, ‘of’ : strings, t : types ->
    lists_of_’T_ (t) : types,
x : ?a, ‘:’ : strings, ‘:’ : strings, y : lists (?a) ->
    ’A_:_:_’L_ (x, y) : lists (?a).
```

Before parsing, the input string is split on all spaces and on all symbols that are not letters, except for digits or letters connected to words with `_` or `^`. For example the string `var10 *x_11* of: ?a^2` would be split into `var, 1, 0, *, x_11, *, of, :, ?, a^2`. Then, each part is encoded as a term of type strings and we apply the extended rewrite rules derived from syntax definitions to the sequence of string terms decoded from the input string.

One can think about an algorithm for applying these extended rewrite rules as an extension of bottom up parsing of context free grammars. In our case, however, the rules include polymorphic types and not just a finite set of non-terminals, so it is more complex to apply them everywhere, and at the same time more things can be expressed easily in this way.

To apply a set of extended rewrite rules to a sequence of terms $t_1, \ldots, t_n$ we will store, for each pair of positions $1 \leq i \leq j \leq n$ in the sequence, all terms $t$ that can be derived between these two positions together with the reconstructing substitutions $\rho'(t)$. These will sometimes extend the previously defined substitutions $\rho(t)$ so that types of rewritten term variables will not be forgotten. We will denote the set of all terms derived between $i$ and $j$ by $d[i, j]$ and we will compute all derivable terms between all positions. We start with $d[i, i] = t_i$ and $d[i, j] = \emptyset$ in other cases and look for a fixed-point of the following extension of the sets $d[i, j]$.

To extend sets of derivable terms, we can take any sequence of positions $1 \leq i_1 \leq i_2 \leq \ldots \leq i_{m+1} \leq n$ and terms $u_k \in d[i_k, i_{k+1}]$ $(k = 1, \ldots, m)$, for which $\rho' = \text{subst}\{\rho'(u_1), \ldots, \rho'(u_m)\}$ exists. Further, we need an extended rewrite rule $l_1, \ldots, l_m \to r$ such that for some term substitution $\sigma$ it holds that $l_k\sigma = u_k$ for all $k$. Then, we can extend the set $d[i_1, i_{m+1}]$ by setting $d[i_1, i_{m+1}] := d[i_1, i_{m+1}] \cup \{r\sigma\}$ with $\rho'(r\sigma) = \rho(r\sigma) \cup \rho'$.

We will continue this process to reach all possible derivable terms for the whole expression and if there is only one term in $d[1, n]$ we will return it as the result. If there are more terms in $d[1, n]$ we will report the *ambiguity* error, and the *no parse* error occurs if $d[1, n] = \emptyset$. It can also happen that the sets will be extended infinitely, but we can prevent such cases using subsumption, which is described below, and with additional rule checking before the parsing starts.

In practice, when there are many extended rewrite rules, we have to first look at what strings come in what order in the input and use only such rules, for which all strings on the left side of the rule can be found in the derived set, and therefore it is possible to apply the rule. In this case, if we derive a

new string for some position then we might have to increase the number of considered rules.

Let us analyze, for example, how the term `x :: Nil` is parsed, where the variable $x$ has type lists($\alpha$). First, we apply twice the rule coming from the definition of variable $x$ that changes the string `'x'` to the term $x$ from lists($\alpha$), and the rule for `'Nil'` to get the term Nil from lists($\alpha'$). Then, we apply the syntax definition of `::` to get the only possible parsing result `Cons (x, Nil)`.

There is one more issue, as if we tried to use to the presented solution in practice we would very often get ambiguity errors. The first thing we have to do to avoid this is to define that a term $t$ with reconstructing substitution $\rho'(t)$ *subsumes* another term $r$ with another substitution $\rho'(r)$ if there is a type substitution $\tau$ such that $\rho'(t)\tau \subseteq \rho'(r)$ and a substitution $\sigma$ such that $t\sigma = r$. In this way we specify when one intermediate result of parsing is more general than another, and we will only consider the most general intermediate results, i.e., between any two positions we will only consider such terms and type substitutions for reconstructed variables that are not subsumed by any other one derivable between these two positions. Considering such subsumptions is especially useful when we have syntax definitions for casts from one type to another, as more and less general types can be often derived in such cases.

Another feature that we have to add is the possibility of defining rule priority, binding strength, and associativity of syntax definitions, in order to parse `1 + 2 + 3 * 4` in a correct way without using parentheses. We can incorporate this into our algorithm in such a way, that we first compute all derivable terms with derivation trees and later we select only the best derivations according to certain priority rules. The priority rules formalize the fact that when an operator $\circ$ is left-associative then $(x \circ y) \circ z$ has a higher priority than $x \circ (y \circ z)$, the converse being true for right-associative operators, and if one binds stronger than the other then it is respected. To check associativity and binding, we need to rotate the tree representing the term, but if we parse two ambiguous terms then we use rule priorities. If the symbol $f$ has a bigger priority than $g$, then $f(t_1, \ldots, t_n)$ has a bigger priority than $g(r_1, \ldots, r_m)$. When the symbols $f$ and $g$ have the same priority then we can say that $f(t_1, \ldots, t_n)$ is bigger than $g(r_1, \ldots, r_m)$ only if $n \geq m$, $t_1 \geq r_1, \ldots, t_n \geq r_n$, and some $t_i > r_i$. To use casts from one type to another in practice we have to add a special priority and priority comparison rule, which means the following. To compare the cast $cast(t)$, derived using a rule with the special priority, with the term $s$, we have to compare $t$ and $s$ first, and if $t$ turns out to be bigger than $s$ then choose $cast(t)$, otherwise choose the cast-free term $s$. Of course, even with these rules there are many incomparable derivations and we still can get ambiguities, but it is rare in practice.

To make the language context dependent and more flexible we assume that any command sent to the system is first processed by the `preprocess_command` function. When this function is not defined then the command is left without processing but the possibility to define and redefine this function with rewrite rules allows extension of the language. Another important addition to

the simple parsing algorithm presented before is the handling of compound sentences. We let the user define how sentences can be composed, for example

```
sentence1 "and" sentence2
sentence2 "where" sentence1
```

and then, before the parsing begins, we divide the text along these composition rules and parse the first sentence before parsing the next.

Using the presented methods we are not only able to parse complex expressions, which can be used to comfortably write programs in the presented model, but we can also use it for basic natural language processing. We can directly translate FrameNet frames [1] to types in our model and use frame rules for specific lexical units as syntax definitions. Then, many sentences written in natural language will be parsed to terms that denote their grammatical structure, and sometimes also a part of the semantic structure. Then we can define functions operating on these terms and allow interaction with the defined system in natural language. In this way simple programming and some program searches described in the next section can be done by non-programmers, which makes the system usable in practice.

When operating on larger sets of functions, types, constructors, and rewrite rules, it is useful to mark them in some way and to be able to choose the ones that we want to use at a given moment. Therefore, we will assign to each function, type and, constructor a set of tags in the form $key = value$, where both *key* and *value* are strings. As mentioned before, some special tags can also be used to generate additional rules or stop in-lining of functions. We can activate and deactivate all symbols with a given set of tags set to a specific or to any possible value. As we did not yet present the commands necessary for adding rewrite rules, setting priorities, removing type, constructor, and function definitions, rewriting terms and for tags, let us give here a simple example.

```
Define function integers *+* integers into integers
    priority normal associativity left
    with tags [context = arithmetics, system = true].
Let 0 + 0 be 0. Compute 0 + 0.
Remove function arg + arg. Remove class lists of ?a.
Activate with tags [system = ANYTHING].
Deactivate with tags [context = arithmetics].
Close context.
```

Note the `close context` command, which removes all variable definitions and opens a new set of variable names, so we can use the variable named $x$ in different contexts with different types. Also, by removing functions and types we have to check if these are not being used somewhere in other definitions or in the internal database.

### 3.2 Compiling Typed Rewriting Systems

We presented a nice model of computation and showed how to represent programs in a readable form, but we need to have some means of executing the programs. Of course, we could easily write a term rewriting interpreter, but as we expect to work with complex and time consuming programs it is necessary to have a more efficient method to execute them.

In general, it is not difficult to compile term rewrite rules to a functional language with polymorphic types, but the compiled programs might be quite inefficient. One method to improve performance is to make it possible to write function and type definitions in the language to which we compile and then, during compilation, substitute these types and functions by their more efficient hand-written counterparts. In this case we have to duplicate our work and write the same programs both as term rewriting rules and in the language to which we compile and we can make mistakes in the translation. To avoid this we will introduce a few optimizations of the rewriting rules and show how to generate efficient code, so that writing the same pieces of code by hand for efficiency will be necessary only in rare cases or for special very often used system functions and types, like arithmetics or lists.

The best compilation method would be to have a formal model of the target language and to use advanced program search algorithms to find efficient equivalent code. This is not possible at present, because neither are our program analysis methods advanced enough nor is the construction of a simple but credible model of a mainstream programming language easy.

For practical reasons we have to stick to more standard compilation methods. Functional programming languages have been present in academia for a long time, and recently some of the associated ideas started to be used in the industry. Polymorphic types, under the name of "generics", are already included in Java and in C# and there is extensive commercial work going on to construct efficient compilers for polymorphically typed languages.

There is also ample research concerning these issues, for example [23], where list optimizations are presented, but we will show only a few simple optimizations that can be quite easily implemented and perform very well in practice. These focus on improving memory management and increasing the number of tail recursive functions, and are similar to the linear optimization described in [11]. Despite their simplicity and efficiency, such optimizations have not been implemented in widely used compilers, perhaps because they rely heavily on the lack of any side effects during computation, which is true for programs in our model, but uncommon in other models.

Let us first show how to translate rewriting rules to C code using as example the concatenation function defined by the rules:

```
concat (Nil, x) -> x,
concat (x :: y, z) -> x :: concat (y, z).
```

We will discuss a basic translation to C code here to show the ideas, although we find it more practical to use C++ and generate separate classes for each

type. Templates can be used for fast polymorphism and overloading to make the copy and comparison functions work on all terms, even on predefined classes like integers. In this way, it is also easier to handle terms with variables and add meta functionality to the generated code without losing efficiency, as you can generate any special function with a given name for each type, add a default one as a template, and let the C++ compiler handle overloading when the function is used.

It should be clear how to implement term matching with a tree of `if` or `case` expressions, and we will assume that there is a record `term_t` defined in C that stores the id of the symbol at the root position in the term and an array of sub-terms.

We will define the concatenation function in C so that it takes an additional argument, a pointer to a term_t where the result will be stored. So, taking matching into account, the concatenation function in C looks like this:

```
void concat (term_t arg0, term_t arg1, term_t *result)
{
  if (arg0.id == Nil_ID) {
    code for the first rule
  }
  else {
    code for the second rule
  }
}
```

Now we will generate the code for the rules, but we will treat constructors in a different way than function symbols. For a function symbol, we will have to generate the arguments first and store them in the variables, and then call the function, whereas for the constructors we will first allocate them and later continue code generation with changed result pointers. Let us look at how code is generated for the constructor in the second rule, where args0.subterms[0] corresponds to the variable $x$ in the rewrite rule.

```
*result = NEW_TERM (Cons_ID, 2);

code for assigning arg0.subterms[0] to (*result).subterms[0]
code for assigning the other part to (*result).subterms[1]
```

When constructing the code for the other part we will first assign the term `y` to the new variable `x0` and the term `z` to `x1`, and in the last line call the concatenation function with

```
concat (x0, x1, & (*result).subterms[1]);
```

In this way, we managed to use the knowledge about which symbol is a constructor and which is a function symbol to create a tail-recursive version of the concatenation function. We could also remove the necessity to allocate memory for some of the variables by reusing the terms from the left side. Instead of

allocating memory for the result and setting the id with the `NEW_TERM` macro, we could just set the result to `arg0` and then change the id and pointers when necessary. It is easy to reuse memory allocated on the left side and variables if they occur the same number of times on the left and right side of a rewriting rule, but we did not present it here in detail for clarity. Adding the possibility to handle certain types and functions in the external language, e.g. integers directly in C, requires additional work, especially to prevent boxing and unboxing where possible, since then we have to generate a separate version of each polymorphic function for each special type. You can look at [10] for more details about how this can be done and the tradeoffs between time and space efficiency and the size of generated code in this case.

Although we can translate our rewriting system directly to C, we will first do a few optimizations to increase the efficiency of the generated code. The first, quite technical one aims to decrease memory usage and the need to reallocate memory. We will try to make as many rewriting rules linear as possible, so we will try to return all unused arguments. For example, the concatenation function is linear, but the double function

```
double (x) -> Pair (x,x)
```

is not. Not all functions can be made linear, but some optimizations can be done.

We can substitute the functions for which a compound argument is read but only a simple argument is returned by equivalent functions returning also the compound argument. For example, the function that calculates the length of a list should be substituted by a function that calculates the length and returns the list itself.

To clarify the method consider the following example:

```
length (Nil) -> 0
length (x :: xs) -> 1 + length (xs)
argument_length (x) -> (length (x), x)
```

In this case, the argument_length function will have to clone the term $x$ before it can call length, which will in turn destroy its copy of $x$. To avoid this, we could optimize the functions and make length return also the argument it takes, so it becomes equivalent to argument_length. To define it we need a new function increment_append that will operate on an element and a pair and will just do the same what the second rule for length does, but accumulating the unused list.

```
increment_append (x, (n, xs)) -> (1 + n, x :: xs)
length (Nil) -> (0, Nil)
length (x :: xs) -> increment_append (x, length (xs))
```

In this way we are able to improve the efficiency of memory allocation and we can make additional improvements to increase the possibility to reuse constructors, which can further optimize the code.

There is one more important and more semantic optimization we can do. In our model of computation, terms are constructed from well-defined types, so if an argument of a function has a non-variable type we can unfold the function definition by substituting all possible constructors of this type for the argument. For example, in the definition of the list concatenating function we had an argument $y$ from lists($\alpha$) in the rule `concat (x :: y, z)` $\rightarrow$ `x :: concat (y, z)`. Since a list, by the definition of our list constructors, is either an empty list or is constructed from an element and a list, we could substitute these two possibilities and get two new rules:

```
concat (x :: Nil, z) -> x :: concat (Nil, z),
concat (x :: (y :: ys), z) -> x :: concat (y :: ys, z).
```

Now the right sides of these rules can be symbolically reduced and we obtain a new definition of concatenation consisting of the following three rules:

```
concat (Nil, x) -> x,
concat (x :: Nil, z) -> x :: z,
concat (x :: (y :: ys), z) -> x :: y :: concat (ys, z).
```

Please note that with this new definition the concatenation function will be called on long lists only half of the times it would be with the old definition. The price here is that we have to do bigger matching to check all three rules, but we can generate optimal `if` trees for the patterns and the compiler on the lower level, in our case the C compiler, can usually optimize them much better than excessive function calls. Also, if there are some auxiliary non-recursive functions called, these calls can sometimes be completely removed in this way, and function calls for specific classes of arguments can also be optimized.

When the definitions are unfolded it is possible that some function calls will occur multiple times. If we represent terms as directed acyclic graphs (DAGs) that have no isomorphic sub-DAGs that are not identical, then such multiple occurrences will be detected and it will be possible to reduce them to one function call. Also, if one function calls two functions in different subterms then we can execute these functions in separate threads. The increased number of rules achieved with the optimization described above can amortize the cost of creating new threads. The possibility of automatically making the program concurrent, which is not practical in imperative programs that have to update the global state of memory, is very important for efficiency as computer systems are getting more and more parallel. Such simple reductions can sometimes speed up the execution by a large factor, and a large number of functional programs is amenable to such optimizations.

## 4 Reasoning Using Games

Creating and understanding proofs is a complex task, and to deeply understand this process and try to do it automatically requires that we build some

model of proofs to think about. In mathematical logic, proofs were depicted as sequences of statements where one statement follows from another. In such a model, it is easy to check if something is a correct proof, but it can be seen even in school that it is very difficult to find a proof of anything. Therefore, we will consider a different, more intuitive representation, where proofs are modeled by games between two players: Eloise, aiming to prove the requested property, and Abelard, who wants to falsify it. The property is proved if Eloise has a winning strategy in the game, i.e., if Abelard loses no matter how he plays. Such games exist for a number of logics and one can find an overview of related results in [5].

In logic games, whenever we see an existential quantifier or a disjunction in the considered formula, then Eloise moves and chooses an element of the structure to substitute for the variable bound by the quantifier, or one component of the disjunction. Conversely, whenever we see a universal quantifier or a conjunction, then Abelard moves and chooses an element or one component of the conjunction. Let us look at a simple example and prove the property *there exists a number that is smaller than* 3 *and there exists a number that is smaller that* 2. Natural numbers are our structure in this case, and this property is a conjunction of two statements:

(1) there exists a number smaller than 3,
(2) there exists a number smaller than 2.

Since we have a conjunction, the first move belongs to Abelard and he chooses (1) or (2). Then it becomes Eloise's turn to move, since in both formulas there is an existential quantifier at the top position. In the first case, she can choose the number 2 and win and in the second case she can choose the number 0 and win. Therefore, Eloise has a winning strategy that can be described as follows: if Abelard chooses option (1) then choose the number 2 and if he chooses option (2) then choose the number 0. Note that if the thesis in the second case were *there exists a number smaller than* 0, there would be no winning strategy for Eloise as she would not be able to choose such a number, and Abelard would win.

It should also be clear that if we wanted to add a simple induction into this game we could allow the players to substitute a quantified variable $x$ only by 0 or $x+1$. If we try to do this with more quantifiers, problems will arise when we want to induce first on one universally quantified variable, and then on one existentially quantified variable. To solve such problems and capture the whole power of inductive reasoning without losing control over finiteness, we need to redefine the games we use and add a natural number to each position, denoting the *level of visibility* in this position. Then, for each level of visibility we need to define the set of possible actions and for each position on this level we need to assign to each action exactly one outgoing edge in the game graph, and now the players will not just choose moves, but they will choose actions. In this way the state of the play is a word over the alphabet of possible actions, but when the player is at visibility level $i$ we will give him only incomplete

information about the current play – just the letters that come from visibility levels lower or equal to $i$. To say that a player wins such partial information game we can not just present winning strategies but we have to give them stepwise through levels of visibility. Therefore, we require that the winning player first gives her strategy for the first visibility level, then the opponent responds with the strategy for the first level, then the first player gives her strategy for the second level and so on up to the last level.

Games with visibility levels can then finally use a parity or Muller winning condition and capture model checking on (tree,$\omega$)-automatic structures or other reasoning. By other reasoning we mean here especially extensions of the game with syntactic reasoning rules including generalisation or specific rules for quantifier elimination. Note that using existential quantifiers and representing functions with rewrite rules we can use this to search for programs. But, although for games with visibility levels it can be non-trivially hard to determine the winner, one can always win such games using a strategy with finite memory and the winner is always determined. Before we present an extended game for general terms with additional possible moves let us show how logic can be implemented in the discussed rewriting system.

**Logic in the System.** To make it possible to implement logical reasoning in the rewriting system we need to define the type of logical formulas on which we will do the reasoning and also the type of terms so that logic can be represented with meta-rules.

The formulas in our system are defined with respect to a type $T$ of basic terms in equalities in the following way:

(1) if $t$ and $r$ are terms of type $T$ then $t = r$ is a $T$-formula;
(2) if $\varphi$ and $\psi$ are formulas then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \to \psi$ are also formulas;
(3) if $\varphi$ is a formula and $s$ is a string that is meant to be the name of a variable in $\varphi$ then also $\forall s\ \varphi$ and $\exists s\ \varphi$ are formulas.

We can also define the set of free term variables in a formula by $\mathrm{Var}(t = r) = \mathrm{Var}(t) \cup \mathrm{Var}(r)$, $\mathrm{Var}(\varphi \wedge \psi) = \mathrm{Var}(\varphi \vee \psi) = \mathrm{Var}(\varphi) \cup \mathrm{Var}(\psi)$, and $\mathrm{Var}(\forall s\ \varphi) = \mathrm{Var}(\exists s\ \varphi) = \mathrm{Var}(\varphi) \backslash \{s\}$, and we distinguish them from the *bound variables* that appear by quantifiers $\exists$ and $\forall$.

This is the standard type of formulas that we will use and for which we will define reasoning rules, but one can also define other logics with the methods described below. We can store formulas with different attributes, for example their proofs or parts of the proofs, just like any other term in the system database.

To change expressions consisting of terms and types in different ways that are not supported by direct rewriting rules, we need to access them coded on a more direct level. For that purpose, we define in the system the type of terms and the type of types and we specify how terms of any type correspond to coded terms. The coding uses `T_Term` for term constructor, `T_Variable` for term variable, and `TT_Type, TT_Function, TT_Type_var` to encode types. The names of syntax definitions are used as strings in the coding and term

variables are coded together with their types. For example, the term ($x$ : booleans)::[] representing a list with one boolean value can be coded as:

```
T_Term ("'A_:_:_'L_",
    [T_Variable ("x_", TT_Type ("booleans_", []), []),
     T_Term ("[_]_", [])]).
```

To make use of the presented coding we add special functions to the system that allow access to information about already defined types, functions, and recently defined ones. Information about functions is placed in a special type that gives the tags, rewrite rules for the function, and the type of the function; we get analogous data for constructors. To select information in the system we use tag queries that are lists of tag names and optional values. An element with tags satisfies the query if it has tags with the corresponding names defined, and when tag value is given in the query the corresponding element tag must have the same value. The functions

```
get constructors [tag query]
get functions [tag query]
```

retrieve from the system all definitions of constructors or functions that satisfy the given queries. All constructors of a type named $t$ have a special tag `#type = t` for easy access.

To make real use of the described coding we need to transfer functions defined between terms to the normal level, where the functions take arguments and return results of different types. Assume that we have a function that takes $N$ terms and returns a term, $f : \text{term}, \ldots, \text{term} \rightarrow \text{term}$. Then we can define a normal function with given `name` and types `type1`, ..., `typeN` and return type `ret_type` in the following way:

```
Define function [name] [type1] ... [typeN]
    into [res_type] from meta function [f].
```

Such function does what executing $f$ with coding does, i.e., it codes all arguments, executes $f$ on the coded terms and decodes the result back. Additionally, this definition puts into the system new logical formulas that should be proved to guarantee that $f$ indeed has the declared type. These are simple formulas that can be proved just by executing a type checking function. We need to define the type checking function that uses the type and function definitions retrieved from the system to calculate the type of given terms, but this is just a technical problem.

Using meta functions makes it more problematic to compile the functions to C or C++. Sometimes rewriting is requested on terms with variables, so we have to prepare the C++ code for such cases and check it when matching is done. Also, when using built-in C++ types we sometimes have to do boxing to allow term variables to be represented. Additionally, we have to keep in the code the mapping from assigned ids of constructor and function symbols to their string names to be able to execute meta functions.

The possibility of operating on meta level is not trivially implemented, but having it we can define logic and reasoning rules in a clear way. In general we will represent reasoning rules as functions, normally implemented on meta level, that take premises in the form of a $T$-formula and generate conclusions of the same type, usually denoted:

$$\text{premises} : formula \vdash \text{conclusions} : formula.$$

In such functions we can use the information about constructors of a type to make induction on this type, and we can access rewrite rules for specific functions to recognise them and implement specialised decision procedures. Sometimes we need to define new functions or types in the system to be able to construct the conclusions. To make it possible we allow reasoning rules to create a list of system commands that are executed in turn before the rule is evaluated in an analogous way to how compound sentences are parsed.

To give meaning to the reasoning rules we will present a set of basic rules that are assumed to be true, that is they transfer true premises to true conclusions. Formulas proved by means of these rules are also true and we allow extension of the set of rules used for reasoning by bringing proved formulas down to the meta level. More precisely, if we prove a formula $\varphi \to \psi$ using functions $f_1, \ldots, f_n$ and types $t_1, \ldots, t_n$ then we can add a reasoning rule $\delta \wedge \varphi' \vdash \psi'$. In this rule $\varphi'$ and $\psi'$ differ from $\varphi$ and $\psi$ only so that functions and types are replaced with variables with the same type. In $\delta$ we use the possibility to get type constructors and rewrite rules for functions to check that the definitions of all $f_i$ and $t_i$ are equivalent to the definitions of the variables with which these were replaced in $\varphi'$ and $\psi'$. In this way the reasoning rule depends only on the semantic of the functions and types and not on their names.

In the next section we will present the basic reasoning rules and the game used to find proofs and understand them. Using the possibility to construct deduction rules from proved formulas we can prove correctness of logical decision procedures. In this way, decision methods using automata or quantifier elimination can be proved and used when reasoning about corresponding objects. For example, the old procedures for the theory of real numbers with addition and multiplication [22] or for Presburger arithmetics [15] and their modern variants can be implemented.

### 4.1 Reason and Search Game for Terms

Let us now define a game that will allow the search for programs and prove their properties in the typed term rewriting model presented before. Positions in this game are formulas and we assume that free variables are implicitly universally quantified. We will not identify positions that differ only in the names of variables, but their identity will be important in determining the winner if induction is used. In this game, each reasoning rule $\varphi \vdash \psi$ describes

a possible move of Eloise from $\psi$ to $\varphi$ and a possible move of Abelard from $\neg\psi$ to $\neg\varphi$. When we know that $\varphi_1 \vdash \psi, \ldots, \varphi_k \vdash \psi$ and that $\psi \vdash \varphi_1 \vee \ldots \vee \varphi_k$, then we call the set of Eloise's moves from $\psi$ to $\varphi_1, \ldots, \varphi_k$ *complete* and analogous for the moves of Abelard from $\neg\psi$ to $\neg\varphi_1, \ldots, \neg\varphi_k$.

We will also assume that the terms in the equalities inside the positions are always rewritten to their normal forms. When proving properties of functions that do not terminate we might not be able to satisfy this requirement and fall in an infinite loop when trying to normalise a term after a move has been made. We will assume that such moves are disallowed and we will not consider them.

We will first describe the winning condition in the game and give a basic set of simple moves that are sufficient to make induction on the structure of the type as defined by constructors, to generalize the formula, and to substitute parts of the formula using some already proved equalities. To get the full power necessary for all proofs we additionally need to create new reasoning rules as described before or add new types, functions, and prove lemmas. Still, the basic reasoning rules correspond to the notion of a simple proof and should be enough for intuitively easy properties. We also present simple moves that are not complete but can often be used in practice to find the proofs faster.

Note that in many cases the proofs of existence of a function lead to the definition of this function and, therefore, we say that this is also a search game. When functional variables are present we will sometimes add rewrite rules for these functions to the system during the game, or even define new functions during the proof. For existential statements that are proved in a non-constructive way we also allow to define the corresponding functions in the system using a `define ... from formula` command similar to the one we used for meta functions. Of course, such functions can be used only for proofs, they can not be rewritten and compiled, but still sometimes it is useful to have them defined.

Let us first state what positions are trivially winning for Eloise and which for Abelard. The only trivially winning positions for Eloise are the positions $t = t$ for some term $t$, and the positions trivially winning for Abelard are the ones $s_1 = s_2$ where $s_1$ and $s_2$ are ground terms and are not equal.

Of course, if any player can move from a position $p$ to a winning position for her or him, then the position $p$ is also winning, and we will guarantee that each position will be winning for at most one player. When a set of moves is complete then if a player loses with all these moves then she loses in this position. When we use inductive rules we have to check if we get back to a position that is identical to the one from which we started only with new variables. We will discuss this later when inductive moves are presented.

The first kind of moves that we will analyse is very simple; Eloise can move from $\varphi \vee \psi$ and Abelard can move from $\varphi \wedge \psi$ to either $\varphi$ or $\psi$. These moves correspond to the reasoning rule:

$$\varphi \vdash \varphi \vee \psi.$$

For Eloise this is a direct correspondence whereas for Abelard we have to substitute the rule with $\neg\varphi \vdash \neg\varphi \vee \neg\psi$ and remember that $\neg\neg\varphi = \varphi$. For every rule, when we want to extract from it the possible moves of Abelard then we should also remember to substitute negated formulas. The two rules $\varphi \vdash \varphi \vee \psi, \psi \vdash \varphi \vee \psi$ are complete. As we find the moves in the game more intuitive than the reasoning rules used in the implementation, we will stick to presenting the moves.

Another possible move for Abelard at a position $\varphi$ is to try to describe inductively the terms that can be substituted for variables in $\mathrm{Var}(\varphi)$ depending on their type, which is possible to do for $x \in \mathrm{Var}(\varphi)$ by considering all constructors of $\mathrm{type}(x)$ if it is not a type variable and not a functional type. Assume, for example, that there is a position $f(x) = c$ where $x$ is a variable from $\mathrm{lists}(\alpha)$. Then, Abelard can move either to $f(\mathrm{Nil}) = c$ or to $f(\mathrm{Cons}(x_0, x_1)) = c$. All possible constructors from which the type can be constructed must be taken into account and then such a set of moves is complete.

A similar induction is possible for functional variables on the type of any arguments or on the result type, and then new rewrite rules have to be added to the system. For example, let $z$ be a functional variable with type $\alpha, \mathrm{lists}(\alpha) \to \mathrm{lists}(\alpha)$. Then we can make induction on the second argument in such a way that we define a new function in the system named $f_z$ with rewrite rules:

$$f_z(x, \mathrm{Nil}, z_1, z_2) \to z_1(x) \quad , \quad f_z(x, \mathrm{Cons}(y_1, y_2), z_1, z_2) \to z_2(x, y_1, y_2).$$

We have to cover all possible constructors of the chosen argument type and add an appropriate number of new functional variables $(z_1, z_2)$ with passing types. Then we have to replace each occurrence of $z$ by the pair $(z_1, z_2)$ and each call $z(x, y)$ by $f_z(x, y, z_1, z_2)$. When the occurrences of the variable $z$ as a functional value are substituted we will have to change the functions that use it to take the pair $(z_1, z_2)$ as argument instead of $z$ and use $f_z(x, y, z_1, z_2)$ instead of $z(x, y)$. Propagating these changes might require us to define other new functions with appropriate types in the system, but it should be clear that such moves are correct and complete.

When we perform induction on the return type we either set the discussed function to a constant or to one of the variables that has the same type as the result, or we set it to a function call of another function that can use additional intermediate computation results. To make the move we first have to clone the formula in our position to have an appropriate number of functional variables, so instead of $\varphi(z)$ we consider in our example $\varphi(z_1) \vee \varphi(z_2) \vee \varphi(z_3)$ and construct new functions:

$$f_{z_1}(x, y) \to \mathrm{Nil} \quad , \quad f_{z_2}(x, y) \to y \quad , \quad f_{z_3}(x, y, v_1, v_2) \to \ v_1(x, y, v_2(x, y)).$$

Note that the type of the intermediate result $v_2(x, y)$ will be assumed to be as general as possible, so we will take the tuple type for all arguments, and

additionally a string type for other computed information, so finally it will be: pairs(pairs($\alpha$, lists($\alpha$)), strings). Then we replace the variable $z_1$ with $f_{z_1}$, $z_2$ with $f_{z_2}$, and $z_3$ with $f_{z_3}$ in the same way as we did above with $f_z$. In the first formula $\varphi(z_1)$ the variable $z_1$ can disappear because of normalization to Nil, in the second formula it can be substituted by the second argument. In the third one we will now have two new functional variables $v_1, v_2$, and we have to correct the types and perhaps extend the system appropriately to get to the right position $\varphi'() \wedge \varphi''(y) \wedge \varphi'''(c, v_1, v_2)$.

Since we always normalize terms in the positions to which we move, it might happen that during a play we will return to a position in which we have already been, but with different variables. For example, if we have a function $f$ defined by rewrite rules $f(\text{Nil}) \to \text{Nil}$ and $f(\text{Cons}(x, y)) \to f(y)$, then we might want to show that $\varphi = (f(x) = \text{Nil})$ is true. In the position $\varphi$ Abelard must take one of the complete inductive moves described above, so he can either go to $f(\text{Nil}) = \text{Nil}$, which will be rewritten on the fly to $\text{Nil} = \text{Nil}$ and is trivially winning for Eloise, or to $f(\text{Cons}(x_1, x_2)) = \text{Nil}$ which will be rewritten to $f(x_2) = \text{Nil}$, and he could repeat this move infinitely. Eloise can have similar problems trying to prove $\exists x\ f(x) = \text{Cons}(1, \text{Nil})$.

To cope with such issues when a position identical modulo variable renaming is repeated in a cycle or if we have any infinite play we need to be more careful determining who wins. In a simple case when just the position of one player is repeated infinitely often and this player is making an inductive move, then the player loses. But with interleaved existential and universal quantifiers we get a bigger problem. For example, if for some function $g$ we analyse the formula $\exists x\ \forall y\ g(x, y) = \text{T}$ then it can happen that we make in turn induction on $x$ and $y$. But to preserve the meaning of the quantifiers we have to assure that any inductive step for $x$ does not depend on the previous steps for $y$. To guarantee this we might have to consider power-sets of positions and check whether the strategies are correlated there. With more interleaving quantifiers these might even be power-sets of power-sets etc. as the satisfiability problem for automatic structures, which can be reduced to this, has non-elementary complexity in the number of quantifier interleaving occurrences.

There is another important kind of inductive move that Eloise can take and it is also complete. Let us assume we have an equality $f(t_1, \ldots, t_n) = t$ somewhere inside the formula $\varphi$ and that the function $f$ is defined by the set of rewrite rules $l_1 \to r_1, \ldots, l_k \to r_k$. When we say that $f$ is defined by a set of rewrite rules $\mathcal{R}$ we assume that, for any ground terms $u_1, \ldots, u_n$ in normal form, the term $f(u_1, \ldots, u_n)$ can be rewritten at the top position with some rule from $\mathcal{R}$. Moreover, we assume that the order of rule application is not important for rules in the set $\mathcal{R}$. We will assume that functions in our system are exhaustively defined, so the first requirement is satisfied. When we have ordered linear rewrite rules we can always make them independent of the order by enumerating constructors, for example if *and* was defined by $and(\text{T}, \text{T}) \to \text{T},\ and(x, y) \to \text{F}$ then we can change the rules to $and(\text{T}, \text{T}) \to \text{T},\ and(x, \text{F}) \to \text{F},\ and(\text{F}, x) \to \text{F}$ to make them independent of the order.

Let us now return to the equality $f(t_1, \ldots, t_n) = t$ and the rules $l_i \to r_i$ that are exhaustive and do not depend on order, and let $l_i = f(l_i^1, \ldots, l_i^n)$. Since the term $f(t_1, \ldots, t_n)$ will be rewritten by some of these rules when it is substituted to be ground, we can search for the correct rule and substitutions to rewrite it and check the formula later. This corresponds to the possibility for Eloise to move to the position $\psi_1 \vee \ldots \vee \psi_k$ where:

$$\psi_i = \exists \mathrm{Var(l_i)}\ t_1 = l_i^1 \wedge \ldots \wedge t_n = l_i^n \wedge \varphi[f(t_1, \ldots, t_n) = t \leftarrow r_i = t],$$

where $\varphi[f(t_1, \ldots, t_n) = t \leftarrow r_i = t]$ is the position $\varphi$ with the equality $f(t_1, \ldots, t_n) = t$ changed to $r_i = t$. Note that if the position $\varphi$ contains unbound variables the new variables from $l_i$ take the unbound ones as arguments, which corresponds to skolemization.

To clarify, it let us consider a position $\mathit{implies}(t_1, t_2) = \mathrm{F}$ for some terms $t_1$ and $t_2$ with two unbound variables $x$ and $y$, and let $\mathit{implies}$ be a normal implication defined by $\mathit{implies}(\mathrm{F}, v) \to \mathrm{T}$, $\mathit{implies}(\mathrm{T}, \mathrm{T}) \to \mathrm{T}$, $\mathit{implies}(\mathrm{T}, \mathrm{F}) \to \mathrm{F}$. In this case, Eloise can move to the formula:

$$(\exists v\ t_1 = \mathrm{F}\ \wedge\ t_2 = v(x, y)\ \wedge\ \mathrm{T} = \mathrm{F}) \vee$$

$$\vee\ (t_1 = \mathrm{T}\ \wedge\ t_2 = \mathrm{T}\ \wedge\ \mathrm{T} = \mathrm{F}) \vee (t_1 = \mathrm{T}\ \wedge\ t_2 = \mathrm{F}\ \wedge\ \mathrm{F} = \mathrm{F}),$$

which can be winning only for the last component, so Eloise moves to $t_1 = \mathrm{T} \vee \mathrm{t_2} = \mathrm{F}$. Observe that $v$ was a functional variable and took $x$ and $y$ as arguments. This is a complete move and it could be taken inside a quantified formula or a formula with free variables as above. This is not possible, for example, for $\varphi \vee \psi$ as $\forall x\ \varphi \vee \psi \not\Leftrightarrow \forall x\ \varphi \vee \forall x\ \psi$.

As you might have noticed, the induction on functional variables for Abelard will make it possible to prove anything of interest only in very rare cases, as it usually only complicates the problem to induce on functions. But for Eloise it might be very important if she wishes to find a function with a specified property. We made it possible to use intermediate results and added a string type by inducing on the result type of a function to make all computable functions representable in this way, but often we should look for a nicer solution using other functions and types that we already have in the system. Also, when we look for a term with non-functional type it might be useful to represent it as the result of computation of a function that already exists.

More precisely, let us assume that we are looking for a term of type $T$ either to substitute it for a bound variable $x$ or for the result of a function in an inductive move by Eloise for a functional variable. In the second case there are additional parameters $x_1, \ldots, x_n$ that are arguments of the function with $\mathrm{type}(x_i) = T_i$. Let us then take any function $f$ defined in the system with type $S_1, \ldots, S_k \to R$ such that there exists a type substitution $\sigma$ for which $R\sigma = T$ and for some indices $\{i_1, \ldots, i_l\} \subseteq \{1, \ldots, k\}$ we can assign numbers $p(i_m)$ so that $S_{i_m}\sigma = T_{p(i_m)}$. With this function, we can represent the term

we are searching for by $x = f(y_1, \ldots, y_k)$, where for $m \in \{i_1, \ldots, i_l\}$ we have $y_m = x_{p(m)}$, and the other arguments are new variables that we will again be requested to find.

Less formally, we just represent the term we want to find as a function call with any combination of already existing or new arguments. In this way we can use any function from the system that has an appropriate type to find the term we are looking for. Such moves are only optional for Eloise, but in practice it is very common to use the knowledge we have in this way, and many natural problems can be solved in just a few steps if the right functions are known in advance and are used in the right time.

The moves described above form the basis for all proofs and should suffice for very simple properties and to find programs that are not complex. But for even slightly more interesting proofs we need to use other formulas proved before to interact with the one we want to prove. We will present the possible moves for such interaction; these are not complete and some of the formulas used must already be known to be true, winning for Eloise. Keep in mind that we also presented a way to create new reasoning rules when the ones here are not sufficient to solve the problem efficiently.

When Eloise plays in a position $\varphi$ she can choose any term $t$ with type $T$ that appears at some position in some of the equalities in $\varphi$ and has no bound variables, and then move to a position $\psi$ which is identical to $\varphi$ with all occurrences of $t$ at any position in any term in any equality replaced by the variable $x$ with type $T$. We will call this move the generalization of $t$.

To make another move, suppose that we know that a formula $t = s_1 \vee \ldots \vee t = s_n$ is true and we are are in a position $\varphi$ that contains a term $u$ in some equality $u = s$. If, for some position $p$ in $u$ and for some substitution $\sigma$, we have $u|_p = t\sigma$ and no variables in $u|_p$ are bound, then let us define $\psi_i$ to be a position identical to $\varphi$ with the term $u$ replaced by $u[s_i\sigma]_p$. Then we can allow Eloise to move from $\varphi$ to $\psi_1 \vee \ldots \vee \psi_n$.

We allow another way for the players to move or to change the system, which makes it possible to define new types, functions, and construct new positions to analyse using the existing ones as building blocks. These moves are described in a simple way: every player can choose any well typed term, build a well constructed position, and insert it into the game. She can also build a function with arguments and result types that already exist in the system and choose a number of rewrite rules for it. New types can also be constructed by choosing a number of well formed constructors and both for functions and types it is possible to build a few of them at once and make them mutually recursive. It is also possible to prove lemmas and create new reasoning rules. The optional moves combined with the simple ones make it possible to prove complex properties of programs.

As one can see, there is a limited set of sensible basic moves and a wider possibility to make optional moves using the knowledge in the system or creating new types and functions that might be useful later. To play the game in a good way, so that all false formulas are found to be false fast and all

true are proved efficiently, Eloise and Abelard have to use sensible strategies and make appropriate moves according to the situation. Of course, any player strategy that does not skip any infinitely long possible move is a program search procedure, and it will find programs that provably fulfil the specified formula. When the game itself is defined with appropriate type inside the system and possible moves are also defined, we can specify that a strategy is a function that chooses a possible move in a given state of the game, and we can use the game-based search procedure to find better strategies and, therefore, make the strategy self-improve by learning as was described before in the theoretical discussion.

Expressing reasoning as a game makes it possible to understand heuristics that we use for reasoning, like "always look first at a few simple examples before you start to prove" or "do not use one induction after another" as simple strategies in the game. For certain types of positions we can use the decision procedures that already exist, and include them in the game as soon as they are implemented as reasoning rules and proofs of their correctness are given. These procedures do not have to be complex and complete, they can also represent good heuristics. As a very simple example, assume that we are in a position $\exists y\ x + y(x, z) = z$, or there is some more complex arithmetic expression given but only with constants and addition. The first move that any well-acting strategy should take is to substitute $y(x, z) = z - x$ or use an algebraic solver to find the right function to substitute, and in this way incorporate the simple decision procedure into the reasoning game. More powerful reasoning rules using automata and quantifier elimination can also be implemented.

## 5 Conclusions

We showed how program search methods can be used both to solve problems and to automatically construct more efficient problem solvers. After showing a theoretical solution, we demonstrated a convenient model of computation and a game for reasoning. We argued that the model of computation can be practical and efficient and that in our reasoning game we can understand the actions taken and incorporate other decision procedures.

The presented model and reasoning method are both extensive enough to cover the tasks of artificial general intelligence, and simple enough to use them for specific reasoning tasks when programming. We are now working on making the system user-friendly and to build the basic knowledge library for it. It would be valuable to have an extensive standard library of types, programs encoded as rewrite rules, and proofs of important facts about these programs and related types in the presented model. Together with a few simple hand-written heuristics, efficient compilation methods for typed rewrite rules, and the program notation extended to be comfortable to use, this should make it practical to produce proofs of correctness of programs, and even to

generate simple programs automatically. At the same time, it would be an interesting database of tests and formal proofs in a simple theoretical model, so it could be potentially used by other systems and also serve as a set of examples to teach future self-improving procedures. The question of whether it will be achievable to define reasoning heuristics well enough to work for more complex programs directly in the presented model is still open. But, since the moves we take in the reasoning game have clear intuitive meaning, we can hope to formalize our own thinking methods in this way, or, if we fail, at least to understand clearly which of the intuitive steps we take when solving problems are the most problematic ones for AI.

We find that the design of the system that we described and that additionally handles natural language processing forms the basis for AGI. One can not expect things like consciousness or speech recognition from such a system, at least not before they are programmed into it. But one can solve problems and even sometimes write programs automatically only by specifying what properties must hold. This system is also a viable software design and development environment where efficient applications with graphical interfaces can be implemented, and where tedious programming tasks can be automated. As natural language processing is included, one can also ask experts in specific domains to write their knowledge directly into the system and later use this knowledge in programs or for reasoning. When a large base of knowledge and a number of reasoning heuristics are included the system will also be capable to learn from them and optimize its own structure.

Since the presented system manages to establish a correspondence between the natural thinking and language of a human being and formal notation suitable for computers and code generation, it makes the communication and cooperation between people and computers practical in almost any situation when a problem needs to be solved. Therefore, we think that the presented way to bring formal logic together with natural language processing and allow to extend it using numerical heuristics is interesting for future AGI development.

In this chapter we omitted a lot of important related AI research. We did not discuss fuzzy and probabilistic logics and models of computation, although these should certainly be used. Still, we prefer to include them and related verification methods [6] as reasoning procedures in the presented model rather than analyse them as core elements on the same level as the programming primitives and logic. As the topic discussed is very extensive we certainly failed to mention and reference all relevant publications, so for more detailed study you should consult the first four books in the reference list.

## References

1. Atkins BT, Fillmore CJ, *FrameNet*, `www.icsi.berkeley.edu/~framenet/`
2. Baader F, Nipkow T (1998) *Term Rewriting and All That*, Cambridge University Press.

3. Bederson B, *Piccolo Toolkit*, `www.cs.umd.edu/hcil/piccolo/`
4. Gödel K (1931) *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatshefte für Mathematik und Physik 38:173-198.
5. Grädel E (2002) *Model Checking Games*, Proceedings of WOLLIC 02, vol. 67 of Electronic Notes in Theoretical Computer Science, Elsevier.
6. Hurd J (2002) *Formal Verification of Probabilistic Algorithms*, PhD thesis, University of Cambridge.
7. Hutter M (2000) *A Theory of Universal Artificial Intelligence based on Algorithmic Complexity*, Technical Report cs.AI/0004001.
8. Hutter M (2005) *Universal Algorithmic Intelligence: A Mathematical top→down Approach*, this volume.
9. Hutter M (2004) *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*, Springer, Berlin.
10. Kaiser Ł (2003) *Speagram*, `www.speagram.org`.
11. Kirkegaard C (2001) *Borel - A Bounded Resource Language*, Project Report, University of Edinburgh.
12. Kolmogorov AN (1965) *Three Approaches to the Quantitative Definition of Information*, Problems of Information Transmission 1:1-11.
13. Levin LA (1973) *Universal Sequential Search Problems*, Problems of Information Transmission 9(3):265-266.
14. Li M, Vitanyi PMB (1997) *An Introduction to Kolmogorov Complexity and Its Applications*, Springer, Berlin.
15. Presburger M (1929) *Über die Vollständigkeit eines gewissen Systems der Arithmetic ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*, Compte-Rendus dei Congres des Math. des pays slavs.
16. Raskin J (2000) *The Humane Interface: New Directions for Designing Interactive Systems*, Addison-Wesley Professional.
17. Robinson A, Voronkov A, Robinson J (2001) *Handbook of Automated Reasoning*, newblock MIT Press, Cambridge, MA.
18. Schmidhuber J (2002) *Optimal Ordered Problem Solver*, Technical Report IDSIA-12-02.
19. Schmidhuber J (2005) *Gödel Machines: Fully Self-Referential Optimal Universal Self-Improvers*, this volume.
20. Schmidhuber J (2005) *The New AI: General & Sound & Relevant for Physics*, this volume.
21. Solomonoff R (1989) *A System for Incremental Learning Based on Algorithmic Probability*, Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition.
22. Tarski A (1948) *A Decision Method for Elementary Algebra and Geometry*, prepared for publication by JCC Mc Kinsey, U.S. Air Force Project RAND, R-109, the RAND Corporation.
23. Wadler WL (1984) *Listlessness is Better than Laziness*, PhD thesis, Carnegie-Mellon University.