Toss Design and Specification

Toss Team

Contents

1 Specification **5**
- 1.1 Discrete Structure Rewriting . . . 5
- 1.2 Continuous Evolution . . . 6
- 1.3 Logic and Constraints . . . 7
- 1.4 Structure Rewriting Games . . . 8

2 Term Operations **11**
- 2.1 Types . . . 12
  - 2.1.1 Definition of Types . . . 12
  - 2.1.2 Definition of Substitutions . . . 13
  - 2.1.3 Composition of Substitutions . . . 13
  - 2.1.4 Unification of Types . . . 13
  - 2.1.5 Unification Algorithm . . . 14
  - 2.1.6 Internal Type Names . . . 14
- 2.2 Syntax Definitions . . . 17
  - 2.2.1 Automatic Functional Syntax Definitions . . . 17
  - 2.2.2 Generating Names from Syntax Definitions . . . 18
  - 2.2.3 User Interface . . . 19
  - 2.2.4 Basic Syntax Definitions . . . 20
- 2.3 Terms with Types . . . 24
  - 2.3.1 Definition of Terms . . . 24
  - 2.3.2 Definition of Well-Typed Terms . . . 25
  - 2.3.3 Type Reconstruction Algorithm . . . 26
  - 2.3.4 Internal Term Display Format . . . 29
- 2.4 Ground Rewriting . . . 31
  - 2.4.1 Rewrite Rules . . . 31
  - 2.4.2 Application of Rewrite Rules . . . 32
  - 2.4.3 Normalisation . . . 33
- 2.5 Term Simplification . . . 34
  - 2.5.1 Rewriting — choosing an appropriate rewrite rule . . . 34
  - 2.5.2 Normalisation . . . 38
- 2.6 Parser . . . 41
  - 2.6.1 Lexer . . . 41
  - 2.6.2 Definition of the Parser . . . 41

2.6.3 Parsing Algorithm . . . 43
2.6.4 Disambiguation after Parsing . . . 43

3 Algorithms **44**
3.1 UCT Game Playing Algorithm . . . 44
3.2 Type Normal Form . . . 47
3.3 Heuristics from Existential Formulas . . . 48
3.4 Finding Existential Descriptions . . . 49
3.5 Alternative Heuristics with Rule Conditions . . . 51
3.6 Solver Techniques . . . 52

4 Formula and Game Induction **53**
4.1 State Representation and Visual Processing . . . 53
4.2 Logic and Descriptive Complexity . . . 54
4.3 Distinguishing Relational Structures . . . 56
4.3.1 Computing first-order types . . . 57
4.3.2 Guarded types for sparse structures . . . 58
4.3.3 Distinguishing positive and negative structures . . . 59
4.4 Learning Winning Conditions in Games . . . 60
4.5 Learning Legal Moves . . . 61
4.6 Summary of Experimental Results . . . 62

5 GDL to Toss Translation **64**
5.1 Game Description Language . . . 64
5.1.1 Notions Related to Terms . . . 64
5.2 Translation . . . 66
5.2.1 Elements of the Toss Structure . . . 67
5.2.2 Expanding the GDL Game Definition . . . 70
5.2.3 Relations . . . 71
5.2.4 Structure Rewriting Rules . . . 75
5.2.5 Translating Moves between Toss and GDL . . . 80
5.2.6 Translating Formulas and Building Defined Relations . . . 80
5.2.7 Concurrent Moves and Toss Locations . . . 83
5.2.8 Translation-specific Simplification . . . 85
5.3 Game Simplification in Toss . . . 85

6 Implementation **87**

List of Figures

1.1 Rewriting rule and its application to a structure. 6
1.2 Tic-tac-toe as a structure rewriting game. 9

3.1 Evaluation game for tic-tac-toe and a UCT tree. 46

4.1 Relational representation of a 3×3 grid. 53
4.2 A position winning for white and one not winning. 60
4.3 4 positions winning for yellow and 4 not winning. 61
4.4 An illegal pawn move. 62
4.5 A legal and an illegal move in Connect4. 63

5.1 Tic-tac-toe in the Game Description Language. 65

6.1 Dependencies among Toss components. 88

1 Specification

1.1 Discrete Structure Rewriting

To represent a state of our model in a fixed moment of time we use finite relational structures, i.e. labelled directed hypergraphs. A relational structure $\mathfrak{A} = (A, R_1, \ldots, R_k)$ is composed of a universe A and a number of relations $R_1, \ldots, R_k$. We denote the arity of R_i by r_i, so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \ldots, R_k\}$.

The dynamics of the model, i.e. the way the structure can change, is described by *structure rewriting rules*, a generalized form of term and graph rewriting. Extended graph rewriting is recently viewed as the programming model of choice for complex multi-agent systems, especially ones with real-valued components [1]. Moreover, this form of rewriting is well suited for visual programming and helps to make the systems understandable.

In the most basic setting, a rule $\mathfrak{L} \to_s \mathfrak{R}$ consists of two finite relational structures $\mathfrak{L}$ and $\mathfrak{R}$ over the same signature and a partial function $s : \mathfrak{R} \to \mathfrak{L}$ specifying which elements of $\mathfrak{L}$ will be substituted by which elements of $\mathfrak{R}$.

Let $\mathfrak{A}, \mathfrak{B}$ be two structures, τ_e a set of relations symbols to be matched exactly and τ_h a set of relations to be matched only positively. In practice, we also allow some tuples in $\mathfrak{L}$ to be optional; this is a shorthand for multiple rules with the same right-hand side. Optional tuples can also appear in $\mathfrak{R}$ if all elements in the tuple have corresponding elements in $\mathfrak{L}$. In such case, the tuple is included in $\mathfrak{R}$ if and only if a corresponding optional tuple appeared in $\mathfrak{L}$. A function $f : \mathfrak{A} \hookrightarrow \mathfrak{B}$ is a $(\tau_\mathrm{e}, \tau_\mathrm{h})$-*embedding* if f is injective, for each $R_i \in \tau_\mathrm{e}$ it holds that $(a_1, \ldots, a_{r_i}) \in R_i^{\mathfrak{A}} \Leftrightarrow (f(a_1), \ldots, f(a_{r_i})) \in R_i^{\mathfrak{B}}$, and for $R_j \in \tau_\mathrm{h}$ it holds that $(a_1, \ldots, a_{r_j}) \in R_j^{\mathfrak{A}} \Rightarrow (f(a_1), \ldots, f(a_{r_j})) \in R_j^{\mathfrak{B}}$. A $(\tau_\mathrm{e}, \tau_\mathrm{h})$-*match* of the rule $\mathfrak{L} \to_s \mathfrak{R}$ in another structure $\mathfrak{A}$ is an $(\tau_\mathrm{e}, \tau_\mathrm{h})$-embedding $\sigma : \mathfrak{L} \hookrightarrow \mathfrak{A}$. We define the result of an application of $\mathfrak{L} \to_s \mathfrak{R}$ to $\mathfrak{A}$ on the match σ as $\mathfrak{B} = \mathfrak{A}[\mathfrak{L} \to_s \mathfrak{R}/\sigma]$, such that the universe of $\mathfrak{B}$ is given by $(A \setminus \sigma(L)) \cup R$, and the relations as follows. A tuple $(b_1, \ldots, b_{r_i})$ is in the new relation $R_i^{\mathfrak{B}}$ if and only if either it is in the relation in $\mathfrak{R}$ already, $(b_1, \ldots, b_{r_i}) \in R_i^{\mathfrak{R}}$, or there exists a tuple in the previous structure, $(a_1, \ldots, a_{r_i}) \in R_i^{\mathfrak{A}}$, such that for each i either $a_i = b_i$ or $a_i = \sigma(s(b_i))$, i.e. either the element was there before or it was matched and b_i is the replacement as specified by the rule. Moreover, if $R_i \in \tau_\mathrm{e}$ then we require in the second case that at least one b_i was already in the original structure, i.e. $b_i = a_i$. To understand this definition it is best to consider an example, and one is given in

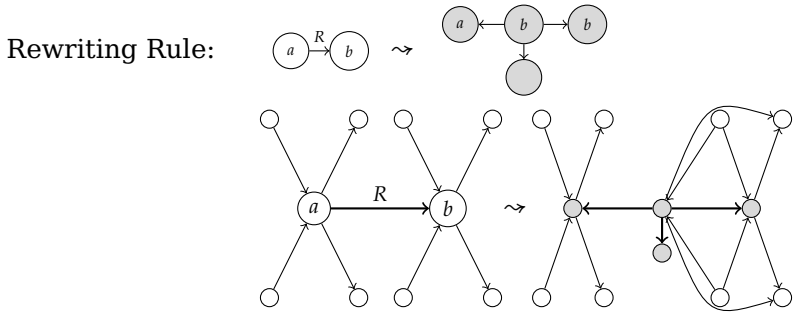


Figure 1.1: Rewriting rule and its application to a structure.

Figure 1.1.

1.2 Continuous Evolution

To model continuous dynamics in our system, we supplement relational structures with a number of labeling functions $f_1, \ldots, f_l$, each $f_i : A \to \mathbb{R}$ (A is the universe).[1] Accordingly, each rewriting rule is extended by a system of ordinary differential equations (ODEs) and a set of right-hand update equations. We use a standard form of ODEs: $f^k_{i,l} = t(f^0_{i,l}, \ldots, f^{k-1}_{i,l})$, where f_i are the above-mentioned functions, l can be any element of the left-hand side structure and f^k denotes the k-th derivative of f. The term $t(\overline{x})$ is constructed using standard arithmetic functions $+, -, \cdot, /$, natural roots $\sqrt[n]{\ }$ for $n > 1$ and rational numbers $r \in \mathbb{Q}$ in addition to the variables $\overline{x}$ and a set of parameters $\overline{p}$ fixed for each rule. The set of right-hand side update equations contains one equation of the form $f_{i,r} = t(\overline{f_{i,l}})$ for each function f_i and each r from the right-hand side structure.

Let $\mathcal{R} = \{(\mathfrak{L}_i \to_{s_i} \mathfrak{R}_i, \mathcal{D}_i, \mathcal{T}_i) \mid i < n\}$ be a set of rules extended with ODEs $\mathcal{D}_i$ and update equations $\mathcal{T}_i$ as described above. Given, for each rule in $\mathcal{R}$, a match σ_i of the rule in a structure $\mathfrak{A}$, the required parameters $\overline{p_i}$ and a time bound t_i, we define the result of a *simultaneous application* of $\mathcal{R}$ to $\mathfrak{A}$, denoted $\mathfrak{A}[\mathcal{R}/\{\sigma_i, t_i\}]$, as follows. We assume that no two intersecting rules have identical time bounds.

First, the structure $\mathfrak{A}$ evolves in a continuous way as given by the *sum* of all equations $\mathcal{D}_i$. More precisely, let $\mathcal{D}$ be a system of differential equations where for each $a \in \mathfrak{A}$ there exists an equation defining $f^k_{i,a}$ if and only if there exists an equation in some $\mathcal{D}_j$ for $f^k_{i,l}$ for some l with $\sigma_j(l) = a$. In such case, the term for $f^k_{i,a}$ is the sum of all terms for such l, with each $f^m_{i,l}$ replaced by the appropriate $f^m_{i,\sigma_j(l)}$. Assuming that all functions f_i and all their derivatives are given at the beginning, there is a unique solution for these variables which satisfies $\mathcal{D}$ and has all other, undefined

[1] In fact $f_i(a)$ is not in $\mathbb{R}$; it is a function $\epsilon \to (x, x + \delta), \delta < \epsilon$.

derivatives set by the starting condition from $\mathfrak{A}$. This solution defines the value of $f_{i,a}(t)$ for each $a \in \mathfrak{A}$ at any time moment t.

Let $t^0 = \min_{i<n} t_i$ be the lowest chosen time bound and let $i_0, \ldots, i_k$ be all rules with this bound, i.e. each $t_{i_m} = t^0$. We apply each of these rules independently[1] to the structure $\mathfrak{A}$ at time t^0. Formally, the relational part of $\mathfrak{A}[\mathcal{R}/\{\sigma_i, t_i\}]$ is equal to $\mathfrak{A}[\mathfrak{L}_{i_0} \to_{s_{i_0}} \mathfrak{R}_{i_0}/\sigma_{i_0}] \cdots [\mathfrak{L}_{i_k} \to_{s_{i_k}} \mathfrak{R}_{i_k}/\sigma_{i_k}]$ and the function values $f_i(a)$ are defined as follows. If the element a was not changed, $a \in \mathfrak{A}$, then we keep the function value from the solution of $\mathcal{D}$, i.e. $f_i(a) = f_{i,a}(t^0)$. In the other case a was on the right-hand side of some rule, $a \in \mathfrak{R}_m$. Let $f_{i,a} = t(\overline{f_{j,l}})$ be the equation in $\mathcal{T}_m$ defining $f_{i,a}$. The new value of $f_i(a)$ is then computed by inserting the appropriate values for $f_{j,l}$ from the solution of $\mathcal{D}$ into $t(\overline{f_{j,l}})$, i.e. $f_i(a) = t(\overline{y_{j,l}})$ where each $y_{j,l} = f_{j,\sigma_m(l)}(t^0)$.

Example. Let us define a simple two-dimensional model of a cat chasing a mouse. The structure we use, $\mathfrak{A} = (\{c, m\}, C, M, x, y)$, has two elements c and m, unary relations $C = \{c\}$ and $M = \{m\}$ used to identify which element is which and two real-valued functions x and y. Both rewriting rules have only one element, both on the left-hand side and on the right-hand side, and the element is in C for the cat rule and in M for the mouse rule. The ODEs for both rules are of the form $x' = p_x, y' = p_y$, where p_x, p_y are parameters. The update equations just keep the left-hand side values, $x_r = x_l, y_r = y_l$. In this setting, simultaneous application of the cat rule with parameters p^c_x, p^c_y for time t^c and the mouse rule with parameters p^m_x, p^m_y for time t^m will have the following effect: The cat will move with speed p^c_x along the x-axis and p^c_y along the y-axis and the mouse analogously with p^m_x and p^m_y, both for time $t^0 = \min(t^c, t^m)$.

1.3 Logic and Constraints

The logic we use for specifying properties of states is an extension of monadic second-order logic with real-valued terms and counting operators. The main motivation for the choice of such logic is *compositionality*: To evaluate a formula on a large structure $\mathfrak{A}$ which is composed in a regular way from substructures $\mathfrak{B}$ and $\mathfrak{C}$ it is enough to evaluate certain formulas on $\mathfrak{B}$ and $\mathfrak{C}$ independently. Monadic second-order logic is one of the most expressive logics with this property and allows to define various useful patterns such as stars, connected components or acyclic subgraphs. (Additional syntactic shorthands can be provided for useful patterns.)

In the syntax of our logic, we use first-order variables ($x_1, x_2, \ldots$) ranging over elements of the structure, second-order variables ($X_1, X_2, \ldots$) ranging over *sets* of elements, fixed-point second order relations ($\Xi_1, \Xi_2, \ldots$) ranging over relations and real-valued variables ($\alpha_1, \alpha_2, \ldots$) ranging over $\mathbb{R}$, and we distinguish boolean formu-

las φ and real-valued terms ρ:

$$\begin{aligned}\varphi &:= R_i(x_1,\ldots,x_{r_i}) \mid x_i = x_j \mid x_i \in X_j \mid \rho <_\epsilon \rho \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \\ &\quad \exists x_i \varphi \mid \forall x_i \varphi \mid \exists X_i \varphi \mid \forall X_i \varphi \mid \exists \alpha_i \varphi \mid \forall \alpha_i \varphi \mid \mathrm{lfp}\Xi_i\varphi \mid \mathrm{gfp}\Xi_i\varphi \\ \rho &:= \quad \alpha_i \mid f_i(x_j) \mid \rho \dot{+} \rho \mid \chi[\varphi] \mid \min_{\alpha_i}\varphi \mid \textstyle\sum_{\overline{x}|\varphi}\rho \mid \prod_{\overline{x}|\varphi}\rho.\end{aligned}$$

Semantics of most of the above operators is defined in the well known way, e.g. $\rho + \rho$ is the sum and $\rho \cdot \rho$ the product of real-valued terms, and $\exists X \varphi(X)$ means that there exists a set of elements S such that $\varphi(S)$ holds, and $\mathrm{lfp}\, X\, \varphi(X)$ is the least fixed-point of the equation $X = \varphi(X)$. Among less known operators, the term $\chi[\varphi]$ denotes the characteristic function of φ, i.e. the real-valued term which is 1 for all assignments for which φ holds and 0 for all other. To evaluate $\min_{\alpha_i}\varphi$ we take the minimal α_i for which φ holds (we allow $\pm\infty$ as values of terms as well). The terms $\sum_{\overline{x}|\varphi}\rho$ and $\prod_{\overline{x}|\varphi}\rho$ denote the sum and product of the values of $\rho(\overline{x})$ for all assignments of elements of the structure to $\overline{x}$ for which $\varphi(\overline{x})$ holds. Note that both these terms can have free variables, e.g. the set of free variables of $\sum_{\overline{x}|\varphi}\rho$ is the union of free variables of φ and free variables of ρ, minus the set $\{\overline{x}\}$. Observe also the ϵ in $<_\epsilon$: the values $f(a)$ are given with arbitrary small but non-zero error and $\rho_1 <_\epsilon \rho_2$ holds only if the upper bound of ρ_1 lies below the lower bound of ρ_2.

The logic defined above is used in structure rewriting rules in two ways. First, it is possible to define a new relation $R(\overline{x})$ using a formula $\varphi(\overline{x})$ with free variables contained in $\overline{x}$. Defined relations can be used on left-hand sides of structure rewriting rules, but are not allowed on right-hand sides. The second way is to add *constraints* to a rule. A rule $\mathfrak{L} \to_s \mathfrak{R}$ can be constrained using three sentences (i.e. formulas without free variables): φ_{pre}, φ_{inv} and φ_{post}. In both φ_{pre} and φ_{inv} we allow additional constants l for each $l \in \mathfrak{L}$ and in φ_{post} special constants for each $r \in \mathfrak{R}$ can be used. A rule $\mathfrak{L} \to_s \mathfrak{R}$ with such constraints can be applied on a match σ in $\mathfrak{A}$ only if the following holds: At the beginning, the formula φ_{pre} must hold in $\mathfrak{A}$ with the constants l interpreted as $\sigma(l)$. Later, during the whole continuous evolution, the formula φ_{inv} must hold in the structure $\mathfrak{A}$ with continuous values changed as prescribed by the solution of the system $\mathcal{D}$ (defined above). Finally, the formula φ_{post} must hold in the resulting structure after rewriting. During simultaneous execution of a few rules with constraints and with given time bounds t_i, one of the invariants φ_{inv} may cease to hold. In such case, the rule is applied at that moment of time, even before $t^0 = \min t_i$ is reached — but of course only if φ_{post} holds afterwards. If φ_{post} does not hold, the rule is ignored and time goes on for the remaining rules.

1.4 Structure Rewriting Games

As you could judge from the cat and mouse example, one can describe a structure rewriting game simply by providing a set of allowed rules for each player. Still, in many cases it is necessary to have more control over the flow of the game and to

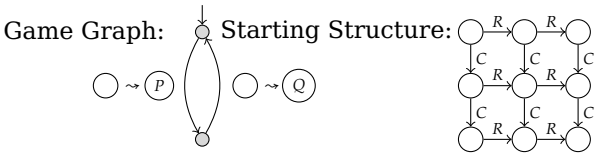


Figure 1.2: Tic-tac-toe as a structure rewriting game.

model probabilistic events. For this reason, we use labelled directed graphs with probabilities in the definition of the games. The labels for each player are of the form:

$$\lambda = (\mathfrak{L} \to_s \mathfrak{R}, \mathcal{D}, \mathcal{T}, \varphi_{\text{pre}}, \varphi_{\text{inv}}, \varphi_{\text{post}}, I_t, \overline{I_p}, \mathrm{m}, \tau_{\mathrm{e}}).$$

Except for a rewriting rule with invariants, the label λ specifies a time interval $I_t \subseteq [0, \infty)$ from which the player can choose the time bound for the rule and, if there are other continuous parameters $p_1, \ldots, p_n$, also an interval $I_{p_j} \subseteq \mathbb{R}$ for each parameter. The element $\mathrm{m} \in \{1, *, \infty\}$ specifies if the player must choose a single match of the rule ($\mathrm{m} = 1$), apply it simultaneously to all possible matches ($\mathrm{m} = \infty$, useful for modeling nature) or if any number of non-intersecting matches might be chosen ($\mathrm{m} = *$); τ_{e} tells which relations must be matched exactly (all other are in τ_{h}).

We define a *general structure rewriting game* with k players as a directed graph in which each vertex is labelled by k sets of labels denoting possible actions of the players. For each k-tuple of labels, one from each set, there must be an outgoing edge labelled by this tuple, pointing to the next location of the game if these actions are chosen by the players. There can be more than one outgoing edge with the same label in a vertex: In such case, all edges with this label must be assigned probabilities (i.e. positive real numbers which sum up to 1). Moreover, an end-point of an interval I_t or I_p in a label can be given by a parameter, e.g. x. Then, each outgoing edge with this label must be marked by $x \sim \mathcal{N}(\mu, \sigma)$, $x \sim \mathcal{U}(a, b)$ or $x \sim \mathcal{E}(\lambda)$, meaning that x will be drawn from the normal, uniform or exponential distribution (these 3 chosen for convenience). Additionally, in each vertex there are k real-valued terms of the logic presented above which denote the payoff for each player if the game ends at this vertex.

A play of a structure rewriting game starts in a fixed first vertex of the game graph and in a state represented by a given starting structure. All players choose rules, matches and time bounds allowed by the labels of the current vertex such that the tuple of rules can be applied simultaneously. The play proceeds to the next vertex (given by the labeling of the edges) in the changed state (after the application of the rules). If in some vertex and state it is not possible to apply any tuple of rules, either because no match is found or because of the constraints, then the play ends and payoff terms are evaluated giving the outcome for each player.

Example. Let us define tic-tac-toe in our framework. The state of the game is represented by a structure with 9 elements connected by binary row and column relations, R and C, as depicted on the right in Figure 1.2. To mark the moves of the players we use unary relations P and Q. The allowed move of the first player is to mark any unmarked element with P and the second player can mark with Q. Thus, there are two states in the game graph (representing which player's turn it is) and two corresponding rules, both with one element on each side (left in Figure 1.2). The two diagonal relations can be defined by $D_1(x,y) = \exists z(R(x,z) \wedge C(z,y))$ and $D_2(x,y) = \exists z(R(x,z) \wedge C(y,z))$ and a line of three by $L(x,y,z) = (R(x,y) \wedge R(y,z)) \vee (C(x,y) \wedge C(y,z)) \vee (D_1(x,y) \wedge D_1(y,z)) \vee (D_2(x,y) \wedge D_2(y,z))$. Using this definitions, the winning condition for the first player is given by $\varphi = \exists x \exists y \exists z(P(x) \wedge P(y) \wedge P(z) \wedge L(x,y,z))$ and for the other player analogously with Q. To ensure that the game ends when one of the players has won, we take as a precondition of each move the negation of the winning condition of the other player.

2 Term Operations

2.1 Types

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

2.1.1 Definition of Types

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

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

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

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

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

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

Let us adopt the notational convention that the type $h_n(R_1, R_2, \ldots, R_n, R)$ will be denoted as $R_1, R_2, \ldots, R_n \to R$. Such types will be called *functional types*. In other words, a type is called a functional type if and only if its top symbol belongs to H. All other types will be called *simple types*.

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

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

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

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

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

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

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

2.1.2 Definition of Substitutions

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

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

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

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

2.1.3 Composition of Substitutions

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

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

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

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

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

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

2.1.4 Unification of Types

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

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

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

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

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

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

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

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

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

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

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

2.1.5 Unification Algorithm

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

2.1.6 Internal Type Names

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

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

Algorithm 1: Algorithm for Computing MGU2

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

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

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

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

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

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

Internal Format Lexer

For certain very technical reasons we need to represent types and terms as strings. Below we present a lexer for dealing with such an internal format.

The lexer is defined by the following delimiters:

(	)	,	[	]	@F	@L	@V	@Y	@T	@:	@‘	@?

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

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

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

2.2 Syntax Definitions

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

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

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

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

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

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

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

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

Examples of syntax definitions

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

2.2.1 Automatic Functional Syntax Definitions

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

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

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

(1) $A = a$

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

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

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

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

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

2.2.2 Generating Names from Syntax Definitions

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

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

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

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

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

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

You should note that in this way names would not be unique and to prevent this we allow a suffix in the generated name that consists of an underscore (_) followed

by an integer followed by a backslash. For example, if we have these two syntax definitions

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

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

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

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

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

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

2.2.3 User Interface

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

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

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

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

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

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

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

2.2.4 Basic Syntax Definitions

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

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

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

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

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

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

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

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

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

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

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

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


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

```
Element ''term'' string ''('' term list '')'' as term.

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

2.3 Terms with Types

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

2.3.1 Definition of Terms

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

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

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

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

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

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

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

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

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

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

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

2.3.2 Definition of Well-Typed Terms

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

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

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

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

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

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

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

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

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

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

2.3.3 Type Reconstruction Algorithm

Type Variable Renaming

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

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

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

(2) $\texttt{Range}(\sigma) \cap A = \varnothing$.

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

The Algorithm

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

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

type symbol	arity
int	0
bool	0
list	1
pair	2

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

And the following variables will be used:

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

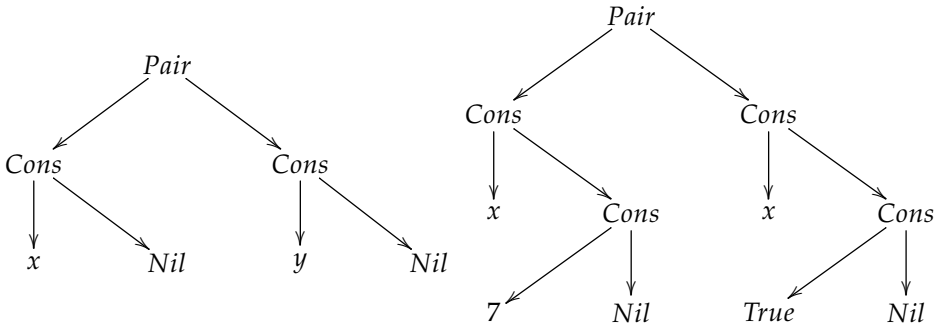
The two examples are:

(1) Pair(Cons(x,Nil),Cons(y,Nil)),

(2) Pair(Cons(x,Cons(7,Nil)),Cons(x,Cons(True,Nil))),

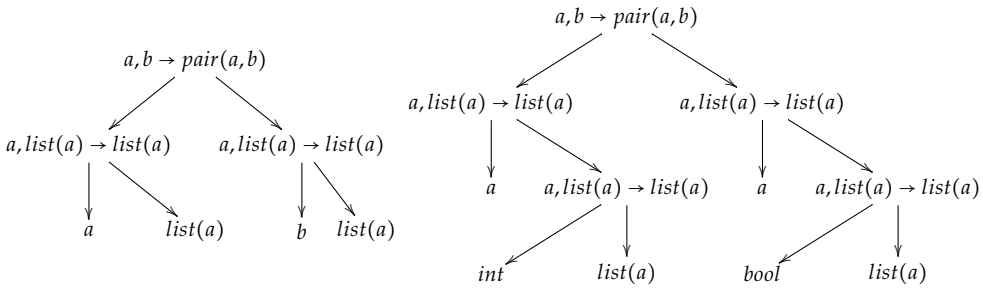
where *vtype* in both cases is such that $vtype(x) = a$ and $vtype(y) = b$.

These terms can be depicted in the following way:



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

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



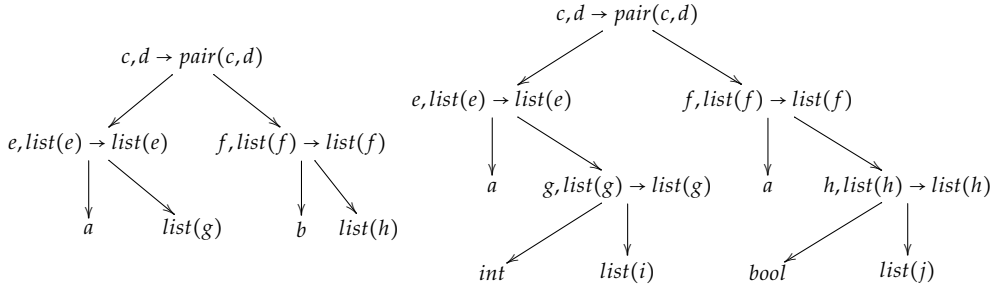
Let `RECOVARS` = `TypeVar`$(vtype($`TermVar`$(t)))) \subset \Theta$. Notice that `RECOVARS` is a finite set of type variables. When the algorithm returns the substitution ρ which reconstructs the types of the term variables in t we will have $Dom(\rho) \subset$ `RECOVARS`.

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

STEP ONE:

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

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



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

Algorithm 2: Type Reconstruction Algorithm: Step Two

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

In the first example, BIGSACK =

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

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

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

In the second example, BIGSACK =

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

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

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

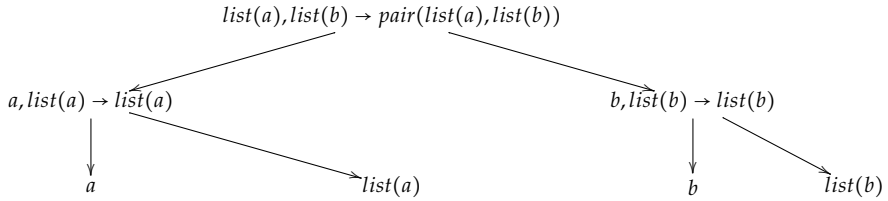
and MGU2(BIGSACK) is empty because ultimately we run against $int \neq bool$.

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

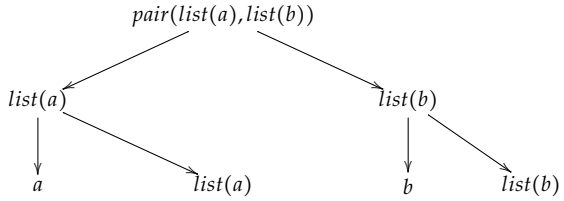
$(t, vtype)$ in that the types of the term variables are now such that there exists a coherent typing for the new term.

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

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



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



2.3.4 Internal Term Display Format

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

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

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

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

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

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

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

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

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

2.4 Ground Rewriting

2.4.1 Rewrite Rules

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

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

Definition of Rewrite Rule

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

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

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

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

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

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

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

Substitution for Term Variables

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

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

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

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

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

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

2.4.2 Application of Rewrite Rules

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

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

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

Rewriting — choosing an appropriate rewrite rule

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

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

No function names and no term variables

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

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

The algorithm for deciding whether a given rewrite rule can be applied to a given term with no function names and no variables is quite straightforward. Visually, let's

have the left-hand side term of the rewrite rule on the left and the term to be rewritten on the right. Now, we traverse the two terms and compare them position by position. If we find a syntactic difference (= different constructors at corresponding positions) we stop and report that the rewrite rule cannot be applied because, naturally, in such a situation there does not exist a substitution required by the definition. And if we find a variable on the left we gather the information that the subtree on the right at the corresponding position should be substituted for this variable. Since this variable can appear multiple times, it can happen that two or more term trees would have to be substituted for this variable. In this case we report that the rule should be rejected. Otherwise, the rule should be applied and we have constructed the necessary substitution in the meantime. (See Algorithm 3.)

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

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

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

2.4.3 Normalisation

We should describe ground normalisation here.

2.5 Term Simplification

We covered ground rewriting in a chapter before. Now we proceed to simlpifying terms where variables might appear.

2.5.1 Rewriting — choosing an appropriate rewrite rule

No function names but term variables allowed

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

Suppose we have these two rewrite rules:

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

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

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

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

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

It is important that the term variables of the left-hand term of the rewrite rule should be disjoint from the term variables of the term to be rewritten. Consider this example: if $f(C(x)) \to h(x)$ is a rewrite rule and $f(x)$ is a term to be rewritten, then in fact we have a clog (= we could potentially apply this rule but we're not sure — after all, the x in $f(x)$ could be of the form $C(z)$ for some z) but according to the definition there is no clog.

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

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

The definition of clog relies on two notions: matching and unification. We have a clog when there is no matching but unification is possible. Unfortunately, if we

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

Description of the Rewriting Algorithm

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

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

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

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

Our approach is practical because checking for unification is too costly and what

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

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

This algorithm returns

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

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

Function names and term variables allowed

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

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

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

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

When we have a term with function names below the top position we must temporarily replace all function calls with term variables in a way which is a one-to-one

Algorithm 4: Rewriting Algorithm

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

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

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

correspondence. Then we treat such a modified term tree with the rewriting algorithm — which will be able to detect a clog in our example — and then we replace the temporary variables back with the original function calls.

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

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

2.5.2 Normalisation

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

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

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

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

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

tion procedure that uses always REWRITE-SPECIAL instead of REWRITE is called NORMALISE-SPECIAL.

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

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

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

Algorithm 5: Normalisation Algorithm — Main Part

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

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

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

In current implementation there is one more special case which prevents function normalisation when a new priority rewrite rule is added. It will probably be changed

in the future so we do not describe it here.

Memoisation

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

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

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

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

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

Of course additionally memoisation needs to be customised. There should be a built-in function `memoise handler [function name as string]` which would return a handler — function that handles memoisation for the given name. The handler should take as arguments the term that is just being rewritten and the number of steps that elapsed and return a ternary truth value — true meaning that we should memoise, false meaning that we should not and unknown meaning that the default system strategy should be applied. We should then recognise if a constant true or false or unknown value is set and then be able to use is efficiently. This will allow fast and efficient switching on and off memoisation for selected functions.

2.6 Parser

2.6.1 Lexer

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

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

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

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

2.6.2 Definition of the Parser

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

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

Additionally, we need a special constructor `term_type_cons_name` and a special function `code_list` which takes a list of typed terms (possibly empty) and returns a typed term which uniquely encodes this list of terms.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.6.3 Parsing Algorithm

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

2.6.4 Disambiguation after Parsing

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

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

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

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

3 Algorithms

3.1 UCT Game Playing Algorithm

When playing a game, players need to decide what their next move is. To represent the preferences of each player, or rather her expectations about the outcome after each step, we use *evaluation games*. Intuitively, an evaluation game is a statistical model used by the player to assess the state after each move and to choose the next action. Formally, an evaluation game $\mathcal{E}$ for $\mathcal{G}$ is just *any* structure rewriting game[1] with the same number of players and with extended signature. For each relation R and function f used in $\mathcal{G}$ we have two symbols in $\mathcal{E}$: R and R_{old}, respectively f and f_{old}.

To explain how evaluation games are used, imagine that players made a concurrent move in $\mathcal{G}$ from $\mathfrak{A}$ to $\mathfrak{B}$ in which each player applied his rule $\mathfrak{L}_i \to_{s_i} \mathfrak{R}_i$ to certain matches. We construct a structure $\mathfrak{C}$ representing what happened in the move as follows. The universe of $\mathfrak{C}$ is the universe of $\mathfrak{B}$ and all relations R and functions f are as in $\mathfrak{B}$. Further, for each $b \in \mathfrak{B}$ let us define the corresponding element $a \in \mathfrak{A}$ as either b, if $b \in \mathfrak{A}$, or as $s_i(b)$, if b was in some right-hand side structure $\mathfrak{R}_i$ and replaced a. The relation R_{old} contains the tuples $\overline{b}$ which replaced some tuple $\overline{a} \in R^{\mathfrak{A}}$. The function $f_{\text{old}}(b)$ is equal to $f_{\text{old}}(a)$ (evaluated in $\mathfrak{A}$) if b replaced a and it is 0 if b did not replace any element. We use $\mathfrak{C}$ as the starting structure for the evaluation game $\mathcal{E}$. This game is then played (as described below) and the outcome of $\mathcal{E}$ is used as an assessment of the move $\mathfrak{C}$ for each player.

As you can see above, the evaluation game $\mathcal{E}$ is used to predict the outcomes of the game $\mathcal{G}$. This can be done in many ways: In one basic case, no player moves in the game $\mathcal{E}$ — there are only probabilistic nodes and thus $\mathcal{E}$ represents just a probabilistic belief about the outcomes. In another basic case, $\mathcal{E}$ returns a single value — this should be used if the player is sure how to assess a state, e.g. if the game ends there. In the next section we will construct evaluation games in which players make only trivial moves depending on certain formulas — in such case $\mathcal{E}$ represents a more complex probability distribution over possible payoffs. In general, $\mathcal{E}$ can be an intricate game representing the judgment process of the player. In particular, note that we can use $\mathcal{G}$ itself for $\mathcal{E}$, but then without evaluation games any more to avoid circularity. This corresponds to a player simulating the game itself as a method to evaluate a state.

[1] In fact it is not a single game $\mathcal{E}$ but one for each vertex of $\mathcal{G}$.

We know how to use an evaluation game $\mathcal{E}$ to get a payoff vector (one for each player) denoting the expected outcome of a move. These predicted outcomes are used to choose the action of player i as follows. We consider all discrete actions of each player and construct a matrix defining a normal-form game in this way. Since we approximate ODEs by polynomials symbolically, we keep the continuous parameters playing $\mathcal{E}$ and get the payoff as a piecewise polynomial function of the parameters. This allows to solve the normal-form game and choose the parameters optimally. To make a decision in this game we use the concept of iterated regret minimization (over pure strategies), well explained in [16].

The regret of an action of one player when the actions of the other players are fixed is the difference between the payoff of this action and the optimal one. A strategy minimizes regret if it minimizes the maximum regret over all tuples of actions of the other players. We iteratively remove all actions which do not minimize regret, for all players, and finally pick one of the remaining actions at random. Note that for turn-based games this corresponds simply to choosing the action which promises the best payoff. In case no evaluation game is given, we simply pick an action randomly and the parameters uniformly, which is the same as described above if the evaluation game $\mathcal{E}$ always gives outcome 0.

With the method to select actions described above we can already play the game $\mathcal{G}$ in the following basic way: Let all players choose an action as described and play it. While we will use this basic strategy extensively, note that, in case of poor evaluation games, playing $\mathcal{G}$ like this would normally result in low payoffs. One way to improve them is the Monte-Carlo method: Play the game in the basic way K times and, from the first actions in these K plays, choose the one that gave the biggest average payoff. Already this simple method improves the play considerably in many cases. To get an even better improvement we simultaneously construct the UCT tree, which keeps track of certain moves and associated confidence bounds during these K plays.

A node in the UCT tree consists of a position in the game $\mathcal{G}$ and a list of payoffs of the plays that went through this position. We denote by $n(v)$ the number of plays that went through v, by $\overline{\mu}(v)$ the vector of average payoffs (for each of the players) and by $\overline{\sigma}(v)$ the vector of square roots of variances, i.e. $\sigma_i = \sqrt{\sum_{p_i}(p_i^2)/n - \mu_i^2}$ if p_i are the recorded payoffs for player i. First, the UCT tree has just one node, the current position, with an empty set of payoffs. For each of the next K iterations the construction of the tree proceeds as follows. We start a new play from the root of the tree. If we are in an internal node v in the tree, i.e. in one which already has children, then we play a regret minimizing strategy (as discussed above) in a normal-form game with payoff matrix given by the vectors $\overline{\mu'}(w)$ defined as follows. Let $\sigma_i'(w) = \sigma_i(w)^2 + \Delta \cdot \sqrt{\frac{2\ln(n(v))}{n(w)}}$ be the upper confidence bound on variance and to scale it let $s_i(w) = \min(1/4, \sigma_i'(w)/\Delta)$, where Δ denotes the payoff range, i.e. the difference between maximum and minimum possible payoff. We set $\mu_i'(w) = \mu_i(w) +$

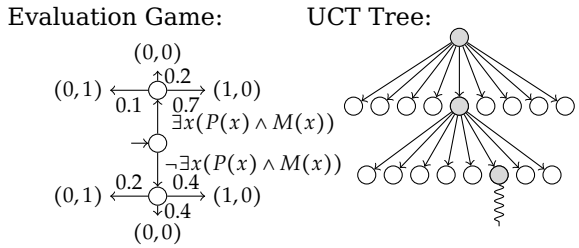


Figure 3.1: Evaluation game for tic-tac-toe and a UCT tree.

$C \cdot \Delta \cdot \sqrt{\frac{\ln(n(v))}{n(w)} s_i(w)}$. The parameter C balances exploration and exploitation and the thesis [9] gives excellent motivation for precisely this formula (UCB1-TUNED). Note that, for turn-based games, when player i moves we select the child w which maximizes $\mu_i'(w)$. When we arrive in a leaf of the UCT tree, we first add all possible moves as its children u and play the evaluation game a few (E) times in each of them. The initial value of $\overline{\mu}$ and $\overline{\sigma}$ is computed from these evaluation plays and we set $n(u) = E'$ ($1 \leq E' \leq E$). After the children are added, we select one and continue to play with the very basic strategy: Only the evaluation game is used to choose actions and the UCT tree is not extended any more in this iteration. When this play of $\mathcal{G}$ is finished, we add the received payoff to the list of recorded payoffs of each node on the played path and recalculate $\overline{\mu}$ and $\overline{\sigma}$. Observe that in each of the K iterations exactly one leaf of the UCT tree is extended and all possible moves from there are added. After the K-th iteration is finished, the action in the root of the UCT tree is chosen taking into account only the values $\overline{\mu}$ of its children.

Example. Consider the model of tic-tac-toe presented previously and let the formula $M(x) = \exists y\, C(x,y) \wedge \exists y\, C(y,x) \wedge \exists y\, R(x,y) \wedge \exists y\, R(y,x)$ express that x is the position in the middle of the board. In Figure 3.1 we depicted a simple evaluation game, which should be interpreted as follows. If the first player made a move to the middle position, expressed by $\exists x(P(x) \wedge M(x))$, then the probability that the first player will win, i.e. of payoff vector $(1,0)$, is 0.7. The probability that the second player will win is 0.1 and a draw occurs with probability 0.2. On the other hand, if the first player did not move to the middle, then the respective probabilities are 0.4, 0.2 and 0.4. When the construction of the UCT tree starts, a payoff vector is assigned to the state after each of the 9 possible moves of the first player. The payoff vector is one of $(1,0), (0,1)$ and $(0,0)$ and is chosen randomly with probabilities $0.7, 0.1, 0.2$ for the middle node in the UCT tree and with probabilities $0.4, 0.2, 0.4$ for all other 8 nodes, as prescribed by the evaluation game. The first iteration does not expand the UCT tree any further. In the second iteration, if the middle node is chosen to play, then its 8 children will be added to the UCT tree. The play in this iteration continues from one of those children, as depicted by the snaked line in Figure 3.1.

3.2 Type Normal Form

To derive evaluation heuristics from payoff terms, we first have to introduce a normal form of formulas which we exploit later in the construction. This normal form is in a sense a converse to the prenex normal form (PNF), because the quantifiers are pushed as deep inside the formula as possible. A very similar normal form has been used recently in a different context [8]. For a set of formulas Φ let us denote by $\mathcal{B}^+(\Phi)$ all positive Boolean combinations of formulas from Φ, i.e. define $\mathcal{B}^+(\Phi) = \Phi \mid \mathcal{B}^+(\Phi) \vee \mathcal{B}^+(\Phi) \mid \mathcal{B}^+(\Phi) \wedge \mathcal{B}^+(\Phi)$.

Definition 1. A formula is in TNF if and only if it is a positive Boolean combination of formulas of the following form

$$\tau = R_i(\overline{x}) \mid \neg R_i(\overline{x}) \mid x = y \mid x \neq y \mid \exists x \mathcal{B}^+(\tau) \mid \forall x \mathcal{B}^+(\tau)$$

satisfying the following crucial constraint: in $\exists x\, \mathcal{B}^+(\{\tau_i\})$ and $\forall x\, \mathcal{B}^+(\{\tau_i\})$ the free variables of *each* τ_i appearing in the Boolean combination *must contain* x.

We claim that for each formula φ there exists an equivalent formula ψ in TNF, and the procedure $\mathtt{TNF}(\varphi)$ computes ψ given φ in negation normal form. Note that it uses sub-procedures `DNF` and `CNF` which, given a Boolean combination of formulas, convert it to disjunctive or respectively conjunctive normal form.

As an example, consider $\varphi = \exists x\big(P(x) \wedge (Q(y) \vee R(x))\big)$; This formula is not in TNF as $Q(y)$ appears under $\exists x$, and $\mathtt{TNF}(\varphi) = \big(Q(y) \wedge \exists x P(x)\big) \vee \exists x\big(P(x) \wedge R(x)\big)$.

Procedure $\mathtt{TNF}(\varphi)$

---

case φ *is a literal* **return** φ;
case $\varphi = \varphi_1 \vee \varphi_2$ **return** $\mathtt{TNF}(\varphi_1) \vee \mathtt{TNF}(\varphi_2)$;
case $\varphi = \varphi_1 \wedge \varphi_2$ **return** $\mathtt{TNF}(\varphi_1) \wedge \mathtt{TNF}(\varphi_2)$;
case $\varphi = \exists x \psi$
  Let $\mathtt{DNF}(\mathtt{TNF}(\psi)) = \bigvee_i (\bigwedge_j \psi_j^i)$
  and $F_i = \{j \mid x \in \mathrm{free}(\psi_j^i)\}$;
  return $\bigvee_i \Big(\bigwedge_{j \notin F_i} \psi_j^i \wedge \exists x (\bigwedge_{j \in F_i} \psi_j^i) \Big)$;
case $\varphi = \forall x \psi$
  Let $\mathtt{CNF}(\mathtt{TNF}(\psi)) = \bigwedge_i (\bigvee_j \psi_j^i)$
  and $F_i = \{j \mid x \in \mathrm{free}(\psi_j^i)\}$;
  return $\bigwedge_i \Big(\bigvee_{j \notin F_i} \psi_j^i \vee \forall x (\bigvee_{j \in F_i} \psi_j^i) \Big)$;

Theorem 2. $\mathtt{TNF}(\varphi)$ *is equivalent to* φ *and in TNF.*

The proof of the above theorem is a simple argument by induction on the structure of the formula, so we omit it here. Instead, let us give an example which explains why it is useful to compute TNF for the goal formulas.

Example 3. As already defined above, the payoff in Tic-tac-toe is given by $\exists x,y,z(P(x) \wedge P(y) \wedge P(z) \wedge L(x,y,z))$. To simplify this example, let us consider the payoff given only by row and column triples, i.e.

$$\varphi = \exists x,y,z\big(P(x) \wedge P(y) \wedge P(z) \wedge \big((R(x,y) \wedge R(y,z)) \vee (C(x,y) \wedge C(y,z))\big)\big).$$

This formula is not in TNF and the DNF of the quantified part has the form $\varphi_1 \vee \varphi_2$, where

$$\varphi_1 = P(x) \wedge P(y) \wedge P(z) \wedge R(x,y) \wedge R(y,z),$$

$$\varphi_2 = P(x) \wedge P(y) \wedge P(z) \wedge C(x,y) \wedge C(y,z).$$

The procedure `TNF` must now choose the variable to first split on (this is discussed in the next section) and pushes the quantifiers inside, resulting in $\texttt{TNF}(\varphi) = \psi_1 \vee \psi_2$ with

$$\psi_1 = \exists x\big(P(x) \wedge \exists y\big(P(y) \wedge R(x,y) \wedge \exists z(P(z) \wedge R(y,z))\big)\big),$$

$$\psi_2 = \exists x\big(P(x) \wedge \exists y\big(P(y) \wedge C(x,y) \wedge \exists z(P(z) \wedge C(y,z))\big)\big).$$

In spirit, the TNF formula is thus more "step-by-step" than the goal formula we started with, and we exploit this to generate heuristics for evaluating positions below.

3.3 Heuristics from Existential Formulas

In this section, we present one method to generate a heuristic from an existential goal formula. As a first important step, we divide all relations appearing in the signature in our game into two sorts, *fluents* and *stable relations*. A relation is called *stable* if it is not changed by any of the structure rewriting rules which appear as possible moves, all other relations are *fluent*. We detect stable relations by a simple syntactic analysis of structure rewriting rules, i.e. we check which relations from the left-hand side remain unchanged on the right-hand side of the rule. It is a big advantage of our formalism in comparison to GDL that stable relations (such as row and column relations used to represent the board) can so easily be separated from the fluents.

After detecting the fluents, our first step in generating the heuristic is to compute the TNF of the goal formula. As mentioned in the example above, there is certain freedom in the `TNF` procedure as to which quantified variable is to be resolved first. We use fluents to decide this — a variable which appears in a fluent will be resolved before all other variables which do not appear in any fluent literal (we choose arbitrarily in the remaining cases).

After the TNF has been computed, we change each sequence of existential quantifiers over conjunctions into a sum, counting how many steps towards satisfying the whole conjunction have been made. Let us fix a factor $\alpha < 1$ which we will discuss later. Our algorithm then changes a formula in the following way.

$$\exists x_1\big(\vartheta_1(x_1) \wedge \exists x_2\big(\vartheta_1(x_2, x_1) \wedge \cdots \wedge \exists x_n\big(\vartheta_n(x_n, \overline{x_i})\big)\cdots\big)\big)$$

$$\rightsquigarrow$$

$$\sum_{x_1 | \vartheta_1(x_1)} \big(\alpha^{n-1} + \sum_{x_2 | \vartheta_2(x_2, x_1)} (\alpha^{n-2} + \cdots (\alpha + \sum_{x_n | \vartheta_n(x_n, \overline{x_i})} 1)\cdots)\big)$$

The sub-formulas $\vartheta_i(x_i, \overline{x})$ are in this case conjunctions of literals or formulas which contain universal quantifiers. The factor α defines how much more making each next step is valued over the previous one. When a formula contains disjunctions, we use the above schema recursively and sum the terms generated for each disjunct.

To compute a heuristic for evaluating positions from a payoff term, which is a real-valued expression in the logic defined above, we simply substitute all characteristic functions, i.e. expressions of the form $\chi[\varphi]$, by the sum generated for φ as described above.

Example 4. Consider the TNF of the simplified goal formula for Tic-tac-toe presented in the previous example and let $\alpha = \frac{1}{4}$. Since the TNF of the goal formula for one player has the form $\psi_1 \vee \psi_2$, we generate the following sums:

$$s_1 = \sum_{x | P(x)} \big(\frac{1}{8} + \sum_{y | P(y) \wedge R(x,y)} (\frac{1}{4} + \sum_{z | P(z) \wedge R(y,z)} 1)\big),$$

$$s_2 = \sum_{x | P(x)} \big(\frac{1}{8} + \sum_{y | P(y) \wedge C(x,y)} (\frac{1}{4} + \sum_{z | P(z) \wedge C(y,z)} 1)\big).$$

Since the payoff is defined by $\chi[\varphi] - \chi[\varphi']$, where φ' is the goal formula for the other player, i.e. with Q in place of P, the total generated heuristic has the form

$$s_1 + s_2 - s_1' - s_2',$$

where s_1' and s_2' are as s_1 and s_2 but with P replaced by Q.

3.4 Finding Existential Descriptions

The method described above is effective if the TNF of the goal formulas has a rich structure of existential quantifiers. But this is not always the case, e.g. in Breakthrough the goal formula for white has the form $\exists x\,(W(x) \wedge \neg\exists y\, C(x,y))$, because

$\neg\exists y\, C(x,y)$ describes the last row which the player is supposed to reach. The general question which presents itself in this case is how, given an arbitrary relation $R(\overline{x})$ (as the last row above), can one construct an existential formula describing this relation. In this section, we present one method which turned out to yield useful formulas at least for common board games.

First of all, let us remark that the construction we present will be done only for relations defined by formulas which do not contain fluents. Thus, we can assume that the relation does not change during the game and we use the starting structure in the construction of the existential formula.

Our construction keeps a set C of conjunctions of stable literals. We say that a subset $\{\varphi_1,\ldots,\varphi_n\} \subseteq C$ *describes* a relation $Q(\overline{x})$ in $\mathfrak{A}$ if and only if Q is equivalent in $\mathfrak{A}$ to the existentially quantified disjunction of φ_i's, i.e. if

$$\mathfrak{A} \vDash Q(\overline{x}) \iff \mathfrak{A} \vDash \bigvee_i \left(\exists \overline{y_i}\ \varphi_i\right),$$

where $\overline{y_i}$ are all free variables of φ_i except for $\overline{x}$.

Our procedure extends the conjunctions from C with new literals until a subset which describes Q is found. These extensions can in principle be done in any order, but to obtain compact descriptions in reasonable time we perform them in a greedy fashion. The conjunctions are ordered by their hit-rank, defined as

$$\texttt{hit-rank}_{\mathfrak{A},Q(\overline{x})}(\varphi) = \frac{|\{\overline{x} \in Q \mid \mathfrak{A} \vDash \exists \overline{y}\varphi(\overline{x})\}|}{|\{\overline{x} \mid \mathfrak{A} \vDash \exists \overline{y}\varphi(\overline{x})\}|},$$

where again $\overline{y} = \texttt{FreeVar}(\varphi) \setminus \overline{x}$. Intuitively, the hit-rank is the ratio of the tuples from Q which satisfy (existentially quantified) φ to the number of all such tuples. Thus, the hit-rank is 1 if φ describes Q and we set the hit-rank to 0 if φ is not satisfiable in $\mathfrak{A}$. We define the $\texttt{rank}_{\mathfrak{A},Q}(\varphi, R(\overline{y}))$ as the maximum of the $\texttt{hit-rank}_{\mathfrak{A},Q}(\varphi \wedge R(\overline{y}))$ and the $\texttt{hit-rank}_{\mathfrak{A},Q}(\varphi \wedge \neg R(\overline{y}))$. The complete procedure is summarized below.

Procedure `ExistentialDescription`$(\mathfrak{A}, Q)$

$C \longleftarrow \{\top\}$
while *no subset of* C *describes* $Q(\overline{x})$ *in* $\mathfrak{A}$ **do**
  for *a stable relation* $R(\overline{y})$, *conjunction* $\varphi \in C$
    with maximal $\texttt{rank}_{\mathfrak{A},Q}(\varphi, R(\overline{y}))$ **do**
      $C \longleftarrow (C \setminus \{\varphi\}) \cup \{\varphi \wedge R(\overline{y}),\ \varphi \wedge \neg R(\overline{y})\}$
  end
end

Since it is not always possible to find an existential description of a relation, let us remark that we stop the procedure if no description with a fixed number of literals is found. We also use a tree-like data structure for C to check the existence of a describing subset efficiently.

Example 5. As mentioned before, the last row on the board is defined by the relation $Q(x) = \neg\exists y\, C(x,y)$. Assume that we search for an existential description of this relation on a board with only the binary row and column relations (R and C) being stable, as in Figure 1.2. Since adding a row literal will not change the hit-rank, our construction will be adding column literals one after another and will finally arrive, on an 3×3 board, at the following existential description: $\exists y_1, y_2(C(y_1,y_2) \wedge C(y_2,x))$. Using such formula, the heuristic constructed in the previous section can count the number of steps needed to reach the last row for each pawn, which is a important e.g. in Breakthrough.

3.5 Alternative Heuristics with Rule Conditions

The algorithm presented above is only one method to derive heuristics, and it uses only the payoff terms. In this section we present an alternative method, which is simpler and uses also the rewriting rules and their constraints. This simpler technique yields good heuristics only for games in which moves are monotone and relatively free, e.g. for Connect5.

Existential formulas are again the preferred input for the procedure, but this time we put them in prenex normal form at the start. As before, all universally quantified formulas are either treated as atomic relations or expanded, as discussed above. The Boolean combination under the existential quantifiers is then put in DNF and, in each conjunction in the DNF, we separate fluents from stable relations. After such preprocessing, the formula has the following form:

$$\exists\overline{x}\left((\vartheta_1(\overline{x}) \wedge \psi_1(\overline{x})) \vee \cdots \vee (\vartheta_n(\overline{x}) \wedge \psi_n(\overline{x}))\right),$$

where each $\vartheta_i(\overline{x})$ is a conjunction of fluents and each $\psi_i(\overline{x})$ is a conjunction of stable literals.

To construct the heuristic, we will retain the stable sub-formulas $\psi_i(\overline{x})$ but change the fluent ones $\vartheta_i(\overline{x})$ from conjunctions to sums. Formally, if $\vartheta_i(\overline{x}) = F_1(\overline{x}) \wedge \cdots \wedge F_k(\overline{x})$ then we define $s_i(\overline{x}) = \chi[F_1(\overline{x})] + \cdots + \chi[F_k(\overline{x})]$, and let $\delta_i(\overline{x}) = F_1(\overline{x}) \vee \cdots \vee F_k(\overline{x})$ be a formula checking if the sum $s_i(\overline{x}) > 0$. The guard for our heuristic is defined as

$$\gamma(\overline{x}) = \left(\psi_1(\overline{x}) \vee \cdots \vee \psi_n(\overline{x})\right) \wedge \left(\delta_1(\overline{x}) \vee \cdots \vee \delta_n(\overline{x})\right)$$

and the heuristic with parameter n by

$$\sum_{\overline{x}|\gamma(\overline{x})\wedge\mathtt{move}(\overline{x})} \left(s_1(\overline{x}) + \cdots + s_n(\overline{x})\right)^n.$$

The additional formula $\mathtt{move}(\overline{x})$ is used to guarantee, that at each element matched to one of the variables $\overline{x}$ it is still possible to make a move. This is done by converting the rewrite rule into a formula with free variables corresponding to the elements of

the left-hand side structure, removing all the fluents F_i from above if these appear negated, and quantifying existentially if a new variable (not in $\overline{x}$) is created in the process. The following example shows how the procedure is applied for Tic-tac-toe.

Example 6. For Tic-tac-toe simplified as before (no diagonals), the goal formula in PNF and DNF reads:

$$\begin{aligned}\exists x,y,z \big(&(P(x) \wedge P(y) \wedge P(z) \wedge R(x,y) \wedge R(y,z)) \\ &\vee (P(x) \wedge P(y) \wedge P(z) \wedge C(x,y) \wedge C(y,z))\big).\end{aligned}$$

The resulting guard is thus, after simplification,

$$\begin{aligned}\gamma(x,y,z) = \big(&(R(x,y) \wedge R(y,z)) \vee (C(x,y) \wedge C(y,z))\big) \\ &\wedge \big(P(x) \vee P(y) \vee P(z)\big).\end{aligned}$$

Since the structure rewriting rule for the move has only one element, say u, on its left-hand side, and $\tau_e = \{P, Q\}$ for this rule, the formula for the left-hand side reads $l(u) = \neg P(u) \wedge \neg Q(u)$. Because P appears as a fluent in γ we remove all occurrences of $\neg P$ from l and are then left with $\texttt{move}(u) = \neg Q(u)$. Since we require that a move is possible from all variables, the derived heuristic for one player with power 4 has the form

$$h = \sum_{x,y,z \mid \gamma(x,y,z) \wedge \neg Q(x) \wedge \neg Q(y) \wedge \neg Q(z)} (\chi[P(x)] + \chi[P(y)] + \chi[P(z)])^4.$$

Since the payoff expression is $\chi[\varphi] - \chi[\varphi']$, where φ' is the goal formula for the other player, we use $h - h'$ as the final heuristic to evaluate positions.

3.6 Solver Techniques

We use a SAT solver (from The Decision Procedure Toolkit, DPT) to operate on symbolic representations of MSO variables. We decided in favor of CNF representation instead of the more standard BDD approach as it seems to scale in a more consistent way.

For handling real arithmetic, we implement a quantifier elimination procedure based on Muchnik's proof. It is not as efficient as CAD (cylindrical algebraic decomposition) but works very consistently for many cases.

The main formula optimization is just performing the TNF, later we only push predicates to the front.

4 Formula and Game Induction

In this chapter we present a method for constructing formulas that separate sets of structures from such sets given as input, and we describe how games can be learned from example plays using this method.

4.1 State Representation and Visual Processing

We represent the state of the game in a fixed moment of time by a finite relational structure, which is the same as a labeled directed hypergraph. Formally, a relational structure $\mathfrak{A} = (A, R_1, \ldots, R_l)$ is composed of a universe A (denoted by the same letter in straight font) and a number of relations. We write r_i for the arity of the relation R_i, so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \ldots, R_l\}$.

Game boards usually have a natural grid-like structure, and to represent them we use relational structures with four binary relations: R for the next-in-a-row relation, C for the next-in-a-column relation, and Da and Db for the two diagonals. The complete structure for the empty 3×3 grid, with 9 elements, is depicted in Figure 4.1. We use this structure to represent the starting position in Tic-Tac-Toe, and larger boards are represented in an analogous way. To mark pieces on the board, we use unary relations (predicates), e.g. a predicate Q for cross and P for circle. In all our experiments we represent game boards in exactly this way, but our learning algorithms work for arbitrary finite relational structures, thus also for more complex scenes and settings.

In the first step, our system reconstructs a sequence of relational structures, representing successive positions in the game, from each input video. We use off-the-

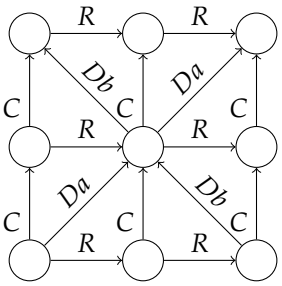


Figure 4.1: Relational representation of a 3×3 grid.

shelf image processing methods in this step, and, as it is not the focus of this work, we describe them only briefly. At the start, we apply the Canny edge detector [2] to the input video stream and use the Hough transform to detect lines in the standard way [4]. We try a few parameters for these operations and use the results to identify the size and position of the board in the frame and to detect the edges of pieces. We mark squares with no edges of pieces as empty, and for the rest we calculate the aggregate color within the edges of the potential piece, adjusted for the aggregate color of all pieces. Finally, based on this adjusted color, we assign the piece to one of the clusters (we used red, blue, yellow and black in the experiments) and mark the grid element with the appropriate predicate in the resulting structure. To determine when a move is made, we use a simple heuristic for hand detection based on the edges detected in the corners of the board.

The above methods for hand and board detection are not perfect and generate false board positions, especially during hand movement. To improve accuracy, we use the fact that only few predicates change in each move. We mark each board with more than two changed predicates as possibly-wrong. A sequence of frames with either a detected hand or a possibly-wrong board represents changes made on the board and is ignored, as we are interested only in the legal positions between such sequences. Among the frames between such sequences, we use majority voting to determine the one configuration of the board all these frames represent. The whole procedure was implemented using the OpenCV library for Canny edge detection and Hough transform and turned out to be sufficient to correctly reconstruct the plays of all games in our experiments.

4.2 Logic and Descriptive Complexity

Before we show how to derive interesting patterns from the sequences of structures reconstructed by the above procedure, we need to introduce some notions from descriptive complexity theory. This section presents the background necessary for this paper, refer to Chapter 3 of [12] for a more complete introduction.

Recall that formulas of first-order logic over a relational signature $\{R_1, \ldots, R_l\}$ and with variables $x_1, x_2, \ldots$ ranging over elements of the structure have the form $\varphi :=$

$$R_i(x_1, \ldots, x_{r_i}) \mid x_i = x_j \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x_i\, \varphi \mid \forall x_i\, \varphi,$$

and their semantics, given an assignment of the variables x_i to elements e_i of the structure, is defined in the natural way, e.g. $\exists x_1 R(x_1, x_2)$ holds for an assignment $x_2 \to e_2$ in a structure $\mathfrak{A}$ if, and only if, there exists an element e_1 such that (e_1, e_2) is in the relation R in $\mathfrak{A}$. Notice that, for the grid structure presented in Figure 4.1, the formula $\neg\exists y\, C(y, x)$ holds exactly for elements from the bottom row. This formula, or the equivalent one $\forall y \neg C(y, x)$ in negation normal form, is a part of the winning condition in games where the goal of one of the players is to reach the bottom row.

First-order logic has several drawbacks from the computational point of view. First of all, it is not expressive enough to describe many relations that can easily be computed. This limitation stems from *locality* of first-order formulas. Intuitively, assume that a neighborhood of an element e in a structure consists of all elements connected to e by any of the relations, and a radius r neighborhood allows r-step connections. Then, a first-order formula can only detect whether certain patterns are present in the structure or not in neighborhoods of a fixed radius. This property, made precise in the theorems of Gaifman [7] and Hanf [17], implies that many patterns cannot be defined in FO.

Example 7. In our representation of the board (Figure 4.1) we allowed only the next-in-a-row relation R. Can we check that two elements are in the same row, but not necessarily next to each other? Indeed, for the 3×3 grid the formula $R(x,y) \vee \exists z(R(x,z) \wedge R(z,y))$ checks that x is left of y on the same row. But already a more complex formula is needed for a 4×4 grid, and the locality theorems imply that there is no FO formula expressing this property on all grids.

To remove this limitation of first-order logic, one extends FO by the *transitive closure operator* that allows to write formulas of the form $\mathrm{TC}\, x,y\, \varphi(x,y)$, which stands for the transitive and reflexive closure of the relation $\varphi(x,y)$. For example, $\mathrm{TC}\, x,y\, R(x,y)$ in our game board representation defines the relation "x is left of y in the same row" we considered above. In this work, we use a more precise operator that specifies exactly the number of steps to take in the transitive closure. Thus, we will write formulas of the form

$$\mathrm{TC}^m\, x,y\, \varphi(x,y),$$

for any first-order formula φ, and the semantics of such a formula is the set of all pairs (x,y) such that one can go from x to y in *exactly* m steps of the relation defined by $\varphi(x,y)$. For example, the formula $\mathrm{TC}^2\, x,y\, R(x,y)$ is equivalent to the first-order formula $\exists z(R(x,z) \wedge R(z,y))$.

Adding the transitive closure operator removes some limitations of FO, but how can we know what other problems remain? To answer this question, one can compare the expressive power of the resulting logic with computational complexity classes. For example, is every polynomial-time computable relation definable in the transitive closure logic, or some other extension of FO? Such questions are studied in *descriptive complexity theory*, and here we recall the most prominent known correspondences. The oldest result [6] shows that the class NP is captured by existential second-order logic. More practically, polynomial-time computations are captured by the least fixed-point logic (LFP) when a linear order relation is present [18, 23]. The requirement of a linear order can be weakened when a counting mechanism is added to the logic, and LFP with counting captures P on many classes of structures, such as grids, planar graphs [13] and all classes that exclude a fixed minor [15]. Finally, while LFP is more expressive than the transitive closure logic (TC)

we mentioned, TC captures all problems solvable in non-deterministic logarithmic space on ordered structures [19] and also on only locally (two-way) ordered graphs [5].

The above results show what is the complexity of the patterns one can define in a logic, but they give little information about the complexity of *learning* formulas. Two most natural metrics for a formula are its size and its *quantifier rank*, i.e. the number of nested quantifiers inside a formula. For any logic $\mathcal{L}$, one can thus state the following problem: Given two finite relational structures, find an $\mathcal{L}$-formula of minimal quantifier rank (or size) distinguishing these structures. Unluckily, already for first-order logic the above problem is hard, namely PSPACE-complete [21]. But there is a natural restriction of first-order logic, the k-variable fragment FO^k, for which this problem becomes solvable in polynomial time [14].

The k-variable fragment of FO consists of all formulas that use only the variables $x_1, \ldots, x_k$, both as free ones and under quantifiers. Note that variables under quantifiers can be renamed, e.g. the formula $\exists x_2(R(x_1, x_2) \wedge \exists x_1 C(x_2, x_1))$ belongs to the 2-variable fragment. When restricted to the k-variable fragment, one must ask whether a formula distinguishing two given structures exists at all in this fragment. Maybe the pattern of interest requires more than k variables? Luckily, for many classes of structures, a constant number of variables is sufficient. These include planar graphs, classes of graphs excluding a minor, and several other classes, see [22] for a survey. The structures we use to represent game boards are planar, and therefore also fall into this category. Let us stress that the restriction to a low number of variables is one key reason why our learning algorithms are efficient, and the above results imply that this will still be the case for more complex structures. Thus, we conjecture that the methods we present below will generalize from board games to various other situations.

4.3 Distinguishing Relational Structures

In this section we present our main learning procedure that, given two sets of structures, the positive and the negative ones, returns a formula φ that holds on all positive structures and on none of the negative ones. As motivated above, the returned formula belongs to the k-variable fragment of first-order logic with the TC^m operator. The formula uses the minimal number of variables k, has minimal quantifier rank among k-variable formulas distinguishing the two sets of structures, belongs to the guarded fragment if possible, and is existential if possible (we will explain these notions and illustrate why they help in learning games later). The procedure runs in polynomial time if each input structure belongs to one of the classes mentioned above, in particular always when the input structures are planar. An important part of the procedure is the computation of $\mathcal{L}$-*types* of tuples of elements from the structures.

Definition 8. The $\mathcal{L}$-type of a tuple $\overline{a}$ in a structure $\mathfrak{A}$ is the subset of formulas of $\mathcal{L}$, with as many free variables as $|\overline{a}|$, that are satisfied by $\overline{a}$ in $\mathfrak{A}$, i.e.

$$\mathcal{L}\text{-type}(\mathfrak{A},\overline{a}) = \left\{\varphi(\overline{x}) \in \mathcal{L} \;\middle|\; |\overline{x}| = |\overline{a}| \text{ and } \mathfrak{A} \models \varphi(\overline{a})\right\}.$$

The set described above is most often infinite for trivial reasons, e.g. it might contain formulas $P(x), P(x) \wedge P(x), P(x) \wedge P(x) \wedge P(x)$, and so on – something that could be described just by $P(x)$. Since in many cases there exists one formula describing this set, we will often abuse the terminology and say that the $\mathcal{L}$-type of $\overline{a}$ in $\mathfrak{A}$ is a single formula $\tau \in \mathcal{L}$, denoted $\tau = \mathrm{tp}^{\mathcal{L}}(\mathfrak{A},\overline{a})$, such that:

$$\mathfrak{A} \models \tau(\overline{a}) \text{ and for all } \varphi \in \mathcal{L}\text{-type}(\mathfrak{A},\overline{a}) \text{ holds } \tau(\overline{x}) \Rightarrow \varphi(\overline{x}).$$

Note that, in principle, such a formula τ might not exist in the logic $\mathcal{L}$. But it does exist for fragments of FO that we consider here, e.g. for bounded quantifier rank, bounded number of variables, and for the guarded fragment.

4.3.1 Computing first-order types

For a fixed number of variables k and a bound n on the quantifier rank, let us denote by $\mathcal{L}^{n,k}$ the set of all first-order formulas using only the variables $x_1, \ldots, x_k$, i.e. from the k-variable fragment, and with quantifier rank at most n. Given a structure $\mathfrak{A}$ and a tuple $\overline{a}$ of length k, we will compute the $\mathcal{L}^{n,k}$-type of $\overline{a}$ inductively and denote the result $\mathrm{tp}^{n,k}(\mathfrak{A},\overline{a})$.

For $n = 0$, the formula $\mathrm{tp}^{n,k}(\mathfrak{A},\overline{a})$ is simply a conjunction of all literals satisfied by $\overline{a}$ in $\mathfrak{A}$, which we compute exhaustively. These are often long formulas with few positive atoms, e.g. in the structure in Figure 4.1 the $0,2$-type of the pair of the bottom-left element and the central element is

$$\begin{aligned} &Da(x_1,x_2) \wedge \neg Da(x_1,x_1) \wedge \neg Da(x_2,x_1) \wedge \neg Da(x_2,x_2) \\ &\qquad \wedge \bigwedge_{v,w \in \{x_1,x_2\}} \neg Db(v,w) \wedge \neg C(v,w) \wedge \neg R(v,w). \end{aligned}$$

For $n > 0$, the type $\mathrm{tp}^{n,k}(\mathfrak{A},\overline{a})$ can be computed inductively, as it is given by the following formula:

$$\begin{aligned} \mathrm{tp}^{n-1,k}(\mathfrak{A},\overline{a}) \wedge \bigwedge_{i<|\overline{a}|} \Bigg(&\forall x_i \left(\bigvee_{b \in \mathfrak{A}} \mathrm{tp}^{n-1,k}(\mathfrak{A},\overline{a}[a_i \leftarrow b]) \right) \\ &\wedge \bigwedge_{b \in \mathfrak{A}} \exists x_i \left(\mathrm{tp}^{n-1,k}(\mathfrak{A},\overline{a}[a_i \leftarrow b]) \right) \Bigg), \end{aligned}$$

where $\overline{a}[a_i \leftarrow b]$ denotes the tuple $\overline{a}$ with the i-th element replaced by b. We omit the proof of correctness of this formula here, as it is very similar to the standard proof for FO.

4.3.2 Guarded types for sparse structures

In our procedure, we need to compute the types of all tuples in the structure. Even for 2-variable tuples on an 8×8 grid, this means computing the types for $64^2 = 4096$ tuples, which is slow, and for triples on a 19×19 grid it is not practical any more (though one could do it in parallel on multiple machines). But most of these tuples will be of no use for distinguishing structures because, aside from unary relations, they all have the same type: not connected by any relation. The structures we use to represent boards are *sparse* and thus, in almost all practical cases, at least one distinguishing tuple will be connected by some binary relations in the structure. This property has also been studied and exploited in logic – the fragment of first-order logic that requires tuples to be connected is called the *guarded fragment* and it is the main reason why modal and description logics enjoy good algorithmic properties [11].

Definition 9. The guarded fragment of FO is defined inductively as a syntactic subset given by the following grammar.

$$\begin{aligned}\varphi \ ::= \ & R_i(x_1, \ldots, x_{r_i}) \mid x = x \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \\ & \exists \overline{y}\big(R_i(\overline{x}, \overline{y}) \wedge \varphi(\overline{x}, \overline{y})\big) \mid \forall \overline{y}\big(\neg R_i(\overline{x}, \overline{y}) \vee \varphi(\overline{x}, \overline{y})\big),\end{aligned}$$

where $\varphi(\overline{x}, \overline{y})$ means that *all* free variables of φ must be included in the set $\{\overline{x}\} \cup \{\overline{y}\}$.

Example 10. Formulas of modal logic translate to guarded first-order logic formulas with two variables. For example, a formula with one free variable x_1 expressing "every C-successor of x_1 has an R-successor in which P holds" can be written in the guarded fragment with two variables as:

$$\forall x_2\big(C(x_1, x_2) \to \exists x_1(R(x_2, x_1) \wedge P(x_1))\big).$$

Again, for a fixed number of variables k and a bound n on quantifier rank, we denote by $\mathcal{G}^{n,k}$ the set of all guarded first-order formulas using only the variables $x_1, \ldots, x_k$, i.e. from the k-variable fragment, and with quantifier rank at most n. Given a structure $\mathfrak{A}$ and the tuple $\overline{a}$ of length k, we will compute the $\mathcal{G}^{n,k}$-type of $\overline{a}$ inductively and denote the result $\mathrm{tp}_G^{n,k}(\mathfrak{A}, \overline{a})$.

For $n = 0$ we have $\mathrm{tp}_G^{0,k}(\mathfrak{A}, \overline{a}) = \mathrm{tp}^{0,k}(\mathfrak{A}, \overline{a})$ as there is no difference between full and guarded logic.

For $n > 0$, the construction is different: Instead of quantifying over one variable, we find sets $\times$ of variables which can be used in a guard, and quantify over those variables. To this end, we first need to compute all *guarded substitutions* of the tuple $\overline{a}$. We say that $\overline{b}$ is a guarded substitution of $\overline{a}$ if $|\overline{b}| = |\overline{a}|$ and the following holds: There exists a subset $\{b_1, \ldots, b_k\}$ of $\overline{b}$ such that $(b_1, \ldots, b_k) \in R_i$ for some R_i, at least one $b_i \in \overline{a}$, and on all positions $j < |b|$ either $b[j] = a[j]$ or $b[j] = b_i$ for some i.

Let now S be the set of all guarded substitutions of $\overline{a}$ and V the set of all proper subsets of variables $x_1, \ldots, x_{|a|}$. For each non-empty set $\mathsf{x} \in V$ let G_x denote proper guards for x, i.e. formulas $R(\overline{x}, \overline{y})$ such that $\{\overline{x}\} = \mathsf{x}$ and $\overline{y}$ is not empty. For each such $g \in G_\mathsf{x}$ let us denote by S_g the subset of S for which the guard g holds, $S_g = \{\overline{b} \in S \mid \mathfrak{A} \models g(\overline{b})\}$. We define the next guarded type for x and $g \in G_\mathsf{x}$ as

$$\begin{aligned}\tau_{\mathsf{x},g} \;=\; & \forall \mathsf{x} \left(g \to \bigvee_{\overline{b} \in S_g} \mathrm{tp}_G^{n-1,k}(\mathfrak{A}, \overline{b}) \right) \\ & \wedge \bigwedge_{\overline{b} \in S_g} \exists \mathsf{x} \left(g \wedge \mathrm{tp}_G^{n-1,k}(\mathfrak{A}, \overline{b}) \right).\end{aligned}$$

Finally, the guarded type $\mathrm{tp}_G^{n,k}(\mathfrak{A}, \overline{a})$ is given by

$$\mathrm{tp}_G^{n-1,k}(\mathfrak{A}, \overline{a}) \;\wedge \bigwedge_{\mathsf{x} \in V} \bigwedge_{g \in G_\mathsf{x}} \tau_{\mathsf{x},g}.$$

Again, we omit the proof that $\mathrm{tp}_G^{n,k}$ is indeed the $\mathcal{G}^{n,k}$-type, as it follows the above construction in a standard way.

An even more restricted logic than the n,k guarded fragment is the n,k *existential* guarded fragment, denoted $\mathcal{EG}^{n,k}$ and defined as all formulas from $\mathcal{G}^{n,k}$ in negation normal form in which no universal quantifier occurs. The above formulas allow to compute existential guarded types as well, only the whole universally quantified part must be removed.

4.3.3 Distinguishing positive and negative structures

Let P be a set of positive structures to be distinguished from the set N of negative ones. For a fixed logic $\mathcal{L}$, variable number k and quantifier rank n, the $\mathcal{L}^{n,k}$-distinguishing procedure proceeds as follows. First, it computes the set $\mathcal{N}$ of $\mathcal{L}^{n,k}$-types of all tuples in all structures in N. Then, for every structure $\mathfrak{A} \in P$, it finds an $\mathcal{L}^{n,k}$-type $\tau_\mathfrak{A}$ of some tuple $\overline{a}$ in $\mathfrak{A}$ such that $\tau_\mathfrak{A} \notin \mathcal{N}$. The formula $\varphi = \bigvee_{\mathfrak{A} \in P} \tau_\mathfrak{A}$ holds in each structure in P, because of the corresponding disjunct, and in no structure from N, because then some $\tau_\mathfrak{A}$ would belong to $\mathcal{N}$. Therefore φ distinguishes P from N.

The $\mathcal{L}^{n,k}$-distinguishing procedure described above is used iteratively, starting from the smallest k, for each k from the smallest n, and with fixed n and k starting from the weakest logic: first the existential guarded fragment, then the full guarded fragment, and only finally the full k-variable fragment with quantifier rank n. Additionally, for each k, after the atomic $0,k$-types τ have been computed we check whether, for some m, the formula $\mathrm{TC}^m\, x_1, x_2\, \tau(x_1, x_2)$ distinguishes P from N. This allows to detect basic transitive relations efficiently. The complete `Distinguish` procedure is summarized below.

The above procedure finds formulas distinguishing P from N with minimal number of variables and minimal quantifier rank, but since the $\mathcal{L}^{n,k}$-distinguishing procedure

Procedure `Distinguish`(P, N)

$k \leftarrow 1$
while *$FO^{0,k}$ does not distinguish P from N* **do**
  Try to distinguish P from N with TC formulas
  for $n = 0, \ldots, k+1$ **do**
    Try to $\mathcal{EG}^{n,k}$-distinguish P from N
    Try to $\mathcal{G}^{n,k}$-distinguish P from N
    Try to $\mathrm{FO}^{n,k}$-distinguish P from N
  end
  $k \leftarrow k+1$
end

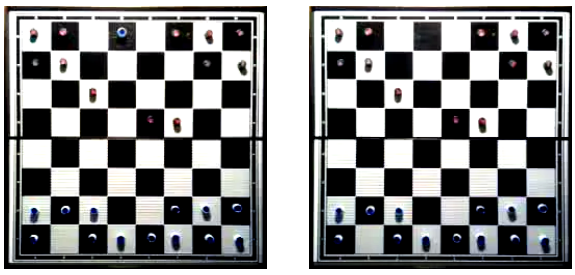


Figure 4.2: A position winning for white and one not winning.

relies on types, the returned formulas are normally very long and hard to read. We correct this by changing the $\mathcal{L}^{n,k}$-distinguishing procedure in the following way. Instead of returning $\varphi = \bigvee_{\mathfrak{A} \in P} \tau_{\mathfrak{A}}$, we will return $\varphi^{\min} = \bigvee_{\mathfrak{A} \in P} \tau_{\mathfrak{A}}^{\min}$, where $\tau_{\mathfrak{A}}^{\min}$ is computed as follows. From all types $\tau_{\mathfrak{A}}^1, \ldots, \tau_{\mathfrak{A}}^l$ which hold for some tuple $\overline{a}$ in $\mathfrak{A}$ but are not in $\mathcal{N}$ (computed previously as well), we greedily remove all literals that are not necessary to distinguish $\mathfrak{A}$ from N. The formula $\tau_{\mathfrak{A}}^{\min}$ is then the shortest of the remaining formulas. In our experiments, it usually turned out to be an easily readable one as well.

4.4 Learning Winning Conditions in Games

The procedure `Distinguish` described above is already sufficient to learn the winning conditions for both players in the games we experimented with. Consider for example the game of Breakthrough in which the goal of the white player is to get to the last row. An example of a winning position is depicted on the left in Figure 4.2, while the same position without the winning piece is on the right. Let $\mathfrak{A}^+$ be the 8×8-grid structure analogous to the one in Figure 4.1 but representing the board on the left in Figure 4.2, and let $\mathfrak{A}^-$ represent the board on the right, with white pieces marked by W and the black ones by B. Running `Distinguish`$(\{\mathfrak{A}^+\}, \{\mathfrak{A}^-\})$

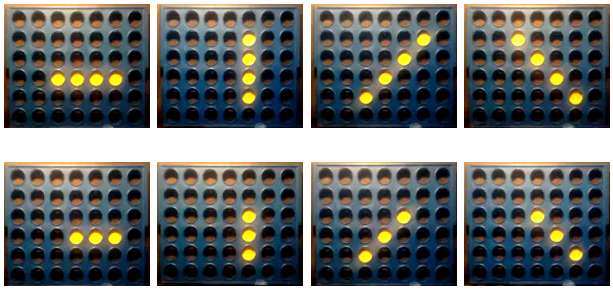


Figure 4.3: 4 positions winning for yellow and 4 not winning.

results in the formula:

$$\exists x_1 (W(x_1) \wedge \forall x_0 \neg C(x_1, x_0)),$$

which expresses that there is a white piece in the last row.

In Figure 4.3, we give another example of 4 positive structures P and 4 negative structures N, this time representing configurations of a 7×6 grid corresponding to winning and not winning positions in Connect4, with Q for yellow. This time, the procedure `Distinguish`(P, N) returns

$$\begin{aligned}\exists x_0, x_1 \Big(& \ \mathrm{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge C(x_0, x_1)\right) \\ & \vee \mathrm{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge Da(x_0, x_1)\right) \\ & \vee \mathrm{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge Db(x_0, x_1)\right) \\ & \vee \mathrm{TC}^3\, x_0, x_1 \left(Q(x_0) \wedge Q(x_1) \wedge R(x_0, x_1)\right)\Big)\end{aligned}$$

as it finds the transitive closures of Boolean combinations of literals distinguishing P from N for $k = 2$ variables.

4.5 Learning Legal Moves

Having learned the winning conditions, we still face the problem of determining which moves are legal and which are not. Since from each video we derive a sequence of structures, and since the underlying grid does not change, we can simply take the symmetric difference of the labels of two successive structures and get a *prototype* of a move: the two sub-structures containing only the elements that changed labels. For example, for Connect4 there would be only 2 prototypes of moves: changing a blank field to a red one, and changing it to a yellow one.

We derive the prototypes of moves from all available sequences, and thus the derived prototypes always cover all presented moves. In some cases, e.g. in Gomoku,

Figure 4.4: An illegal pawn move.

the prototypes are already exactly the desired moves. But in most cases the prototypes are too general, as not every imaginable move is legal. For example, in Connect4 it is not possible to change a blank field to a yellow one or to a red one if there is still a blank field below it. To demonstrate such situations, we also record videos of *illegal* moves, such as the move of a pawn presented in Figure 4.4. Note that a move always consists of two structures.

For every generated move prototype, we gather all pairs of structures in which this move was applied legally, and also all pairs in which it was demonstrated as illegal. From each of these pairs, we take the first structure (the one before the move was applied) and add to it new predicates, marking the elements of the prototype, i.e. the fields on which the predicates change. After such marking, we again have a set of positive structures (the marked first ones from the legal moves) and a set of negative ones. This allows us to again use the `Distinguish` procedure to derive the precondition of a legal move. For example, consider Figure 4.5 in which we present an example of an outcome of a legal and of an illegal move. The field on which the upper red token is placed is the one that changes, so it gets marked by e_1. Let us denote the structures representing these two marked positions – the legal one, depicted on the left in Figure 4.5, and the illegal one, depicted on the right – by $\mathfrak{A}^+$ and $\mathfrak{A}^-$, respectively. Running `Distinguish`$(\{\mathfrak{A}^+\},\{\mathfrak{A}^-\})$ results in the following formula:

$$\exists x_1\big(Q(x_1) \wedge \exists x_0(C(x_1,x_0) \wedge x_0 = e_1)\big),$$

where e_1 is a constant marking the single element of the move prototype. This formula expresses that there must be a yellow element below the changed one.

4.6 Summary of Experimental Results

To learn a complete game, we use four kinds of videos. For the winning conditions, we use videos that present plays ending in positions won by the first player and some with plays ending in positions won by the second player. Additionally, it is

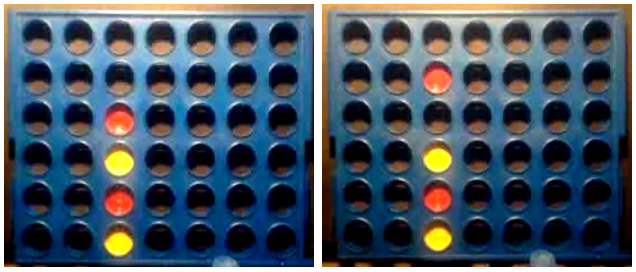


Figure 4.5: A legal and an illegal move in Connect4.

	1 Wins	2 Wins	Not Won	Illegal
Breakthrough	1	1	3	0
Connect4	4	4	13	4
Gomoku	4	4	9	0
Pawn Whopping	1	1	4	6
Tic-Tac-Toe	4	4	17	0

Table 4.1: Number of videos needed for each game.

convenient to allow videos that depict unfinished or tied plays, and, to distinguish legal and illegal moves, we may need illegal move videos. There are therefore 4 possible kinds of videos. The number of videos of each kind that we used to learn the correct rules of each of the example games is given in Table 4.1.

5 GDL to Toss Translation

5.1 Game Description Language

The game description language, GDL, is a variant of Datalog used to specify games in a compact, prolog-like way. The GDL syntax and semantics are defined in [10, 20], we refer the reader there for the definition and will only recapitulate some notions here. When we build first-order formulas over GDL atoms during intermediate steps of translation, they are intended to be interpreted in already existing GDL models (which are defined in [10, 20]).

The state of the game in GDL is defined by the set of propositions true in that state. These propositions are represented by terms of limited height. The moves of the game, i.e. the transition function between the states, are described using Datalog rules — clauses define which predicates hold in the subsequent state. In this way a transition system is specified in a compact way. Additionally, there are 8 special relations in GDL: `role, init, true, does, next, legal, goal` and `terminal`, which are used to describe the game: initial state, the players, their goals, and thus like.

We say that *GDL state terms* are the terms that are possible arguments of `true`, `next` and `init` relations in a GDL specification, i.e. those terms which can define the state of the game. The *GDL move terms* are ground instances of the second arguments of `legal` and `does` relations, i.e. those terms which are used to specify the moves of the players.

The complete Tic-tac-toe specification in GDL is given in Figure 5.1. While games can be formalised in various ways in both systems, Figure 5.1 gives an example of a formalisation in GDL.

5.1.1 Notions Related to Terms

Since GDL is a term-based formalism, we will use the standard term notions, as e.g. in the preliminaries of [3]. We understand terms as finite trees with ordered successors and labelled by the symbols used in the current game, with leafs possibly labelled by variables.

Substitutions. A *substitution* is an assignment of terms to variables. Given a substitution σ and a term t we write $\sigma(t)$ to denote the result of applying σ to t, i.e. of replacing all variables in t which also occur in σ by the corresponding terms. We extend this notation to tuples in the natural way.

```
(role x)
(role o)
(init (cell a a b))
(init (cell b a b))
(init (cell c a b))
(init (cell a b b))
(init (cell b b b))
(init (cell c b b))
(init (cell a c b))
(init (cell b c b))
(init (cell c c b))
(init (control x))
(<= (next (control ?r)) (does ?r noop))
(<= (next (cell ?x ?y ?r)) (does ?r (mark ?x ?y)))
(<= (next (cell ?x ?y ?c)) (true (cell ?x ?y ?c)) (does ?r (mark ?x1 ?y1))
  (or (distinct ?x ?x1) (distinct ?y ?y1)))
(<= (legal ?r (mark ?x ?y)) (true (control ?r)) (true (cell ?x ?y b)))
(<= (legal ?r noop) (role ?r) (not (true (control ?r))))
(<= (goal ?r 100) (conn3 ?r))
(<= (goal ?r 50) (role ?r) (not exists_line3))
(<= (goal x 0) (conn3 o))
(<= (goal o 0) (conn3 x))
(<= terminal exists_line3)
(<= terminal (not exists_blank))
(<= exists_blank (true (cell ?x ?y b)))
(<= exists_line3 (role ?r) (conn3 ?r))
(<= (conn3 ?r) (or (col ?r) (row ?r) (diag1 ?r) (diag2 ?r)))
(<= (row ?r)
  (true (cell ?a ?y ?r)) (nextcol ?a ?b)
  (true (cell ?b ?y ?r)) (nextcol ?b ?c)
  (true (cell ?c ?y ?r)))
(<= (col ?r)
  (true (cell ?x ?a ?r)) (nextcol ?a ?b)
  (true (cell ?x ?b ?r)) (nextcol ?b ?c)
  (true (cell ?x ?c ?r)))
(<= (diag1 ?r)
  (true (cell ?x1 ?y1 ?r))
  (nextcol ?x1 ?x2) (nextcol ?y1 ?y2)
  (true (cell ?x2 ?y2 ?r))
  (nextcol ?x2 ?x3) (nextcol ?y2 ?y3)
  (true (cell ?x3 ?y3 ?r)))
(<= (diag2 ?r)
  (true (cell ?x1 ?y5 ?r))
  (nextcol ?x1 ?x2) (nextcol ?y4 ?y5)
  (true (cell ?x2 ?y4 ?r))
  (nextcol ?x2 ?x3) (nextcol ?y3 ?y4)
  (true (cell ?x3 ?y3 ?r)))
(nextcol a b)
(nextcol b c)
```


Figure 5.1: Tic-tac-toe in the Game Description Language.

MGU. We say that a tuple of terms $\bar{t}$ is *more general* than another tuple $\bar{s}$ of equal length, written $\bar{s} \leq \bar{t}$, if there is a substitution σ such that $\bar{s} = \sigma(\bar{t})$. Given two tuples of terms $\bar{s}$ and $\bar{t}$ we write $\bar{s} \doteq \bar{t}$ to denote that these tuples *unify*, i.e. that there exists a substitution σ such that $\sigma(\bar{s}) = \sigma(\bar{t})$. In such case there exists a most general substitution of this kind, and we denote it by $\mathrm{MGU}(\bar{s}, \bar{t})$.

Paths. A *path* in a term is a sequence of pairs of function symbols and natural numbers denoting which successor to take in turn, e.g. $p = (f,1)(g,2)$ denotes the second child of a node labelled by g, which is the first child of a node labelled by f. For a term t we write $t\downarrow_p$ to denote the subterm of t at path p, and that t has a path p, i.e. that the respective sequence of nodes exists in t with exactly the specified labels. Using $p = (f,1)(g,2)$ as an example, $f(g(a,b),c)\downarrow_p = b$, but $g(f(a,b),c)\downarrow_p$ is false. Similarly, for a formula φ, we write $\varphi(t\downarrow_p)$ to denote that t has path p and the subterm $r = t\downarrow_p$ satisfies $\varphi(r)$. A path can be an empty sequence ϵ and $t\downarrow_\epsilon = t$ for all terms t.

For any terms t, s and any path p existing in t, we write $t[p \leftarrow s]$ to denote the result of *placing s at path p in t*, i.e. the term t' such that $t'\downarrow_p = s$ and on all other paths q, i.e. ones which neither are prefixes of p nor contain p as a prefix, t' is equal to t, i.e. $t'\downarrow_q = t\downarrow_q$. We extend this notation to sets of paths as well: $t[P \leftarrow s]$ places s at all paths from P in t.

5.2 Translation

In this section, we describe our main construction. Given a GDL specification of a game G, which satisfies the restrictions described elsewhere, we construct a Toss game $T(G)$ which represents exactly the same game. Moreover, we define a bijection μ between the moves possible in G and in $T(G)$ in each reachable state, so that the following correctness theorem holds.

Theorem 11 (Correctness)**.**
Let S be any state of G reached from the initial one by a sequence of moves $m_1 \dots m_n$. We write $\mu(S)$ for the state of $T(G)$ reached by $\mu(m_1) \dots \mu(m_n)$. The following conditions are satisfied.

- *The function μ defines a bijection between the moves possible in S and in $\mu(S)$ for each player.*
- *If no move is possible in S (and in $\mu(S)$), then the payoffs in G evaluate to the same value as those in $T(G)$.*

We will not prove this theorem here, but the construction presented below should make it clear why the exact correspondence holds. For the rest of this section let us fix the GDL game specification G we will translate. We begin by transforming G itself: eliminating variables clearly referring to players (i.e. arguments of positive

`role` atoms, first arguments to positive `does` atoms and to `legal`) by substituting them by players of G (i.e. arguments of `role` facts), duplicating the clauses. From this specification, we derive the elements (Section 5.2.1) and the stable relations and initial fluents (Section 5.2.3) of the Toss structure. Having separated the fluent from the stable part of state terms, we further transform the definition G by expanding variables corresponding to fluents, and use the transformed specification to derive the defined relations in Toss, the rewriting rules (Section 5.2.4) and finally the move translation function (Section 5.2.5).

5.2.1 Elements of the Toss Structure

By definition of GDL, the state of the game is described by a set of propositions true in that state. Let us denote by $\mathcal{S}$ the set of all GDL state terms which are true at some game state reachable from the initial state of G.

For us, it is enough to approximate $\mathcal{S}$ from above. To approximate $\mathcal{S}$, we currently perform an *aggregate playout*, i.e. a symbolic play in where all players take all their legal moves in a state. Since an approximation is sufficient, we check only the positive part of the legality condition of each move.

Fluent Paths

We need to decide which parts of the state description will provide the fixed "coordinate system", and which will provide labels over coordinates (predicates ranging over the points spanned by the coordinates). The labels-predicates will be the only means accounting for game state changes, directly translating into Toss fluents, we will therefore call the state term paths containing them the *fluent paths*. We need to have at least one fluent for each `next` clause that leads to state change, but first we need to determine which `next` clauses change state.

We say that a `next` clause $\mathcal{C}$ is a *frame clause* when, for each state transition, each state term it generates is already present in the prior state. If possible we find a frame clause by checking whether it contains a `true` relation applied to a term equal to the `next` argument. Otherwise, we approximate by checking on states generated by a few random playouts.

For each non-frame `next` clause (`<= (next` $s_{\mathcal{C}}$`) ...`), a fluent path is a path p to a leaf in $s_{\mathcal{C}}$ such that the set $\mathcal{S}\downarrow_p = \{t \mid t = s\downarrow_p, s \in \mathcal{S}\}$ is the smallest, where $\mathcal{S}$ is the set of all state terms. When there are several smallest paths, we select p such that $s_{\mathcal{C}} \downarrow p$ is not a variable. We denote the set of all fluent paths by $\mathcal{P}_f$.

Example 12. There are three `next` clauses in Figure 5.1. $\mathcal{C}_1$:

```
(<= (next (cell ?x ?y ?c))
  (true (cell ?x ?y ?c))
  (does ?r (mark ?x1 ?y1))
```

```
  (or (distinct ?x ?x1) (distinct ?y ?y1)))
```

does not lead to any fluent paths, since it is a frame clause. The clause:

```
(<= (next (cell ?x ?y ?r))
  (does ?r (mark ?x ?y)))
```

expands to:

```
(<= (next (cell ?x ?y x))
  (true (control x))
  (true (cell ?x ?y b)))
(<= (next (cell ?x ?y o))
  (true (control o))
  (true (cell ?x ?y b)))
```

These generate the fluent path $(\texttt{cell}, 3)$, since $\mathcal{S}\downarrow_{(\texttt{cell},1)} = \mathcal{S}\downarrow_{(\texttt{cell},2)} = \{a, b, c\}$ and $\mathcal{S}\downarrow_{(\texttt{cell},3)} = \{x, o, b\}$ have the same cardinality, but only $(\texttt{cell}, 3)$ does not point to a variable. The clause:

```
(<= (next (control ?r)) (does ?r noop))
```

expands to:

```
(<= (next (control x))
      (not (true (control x))))
(<= (next (control o))
      (not (true (control o))))
```

These generate the fluent path $(\texttt{control}, 1)$ since they are not frame clauses and $(\texttt{control}, 1)$ points to the only leaf in heads of these clauses. In the end $\mathcal{P}_f = \{(\texttt{cell}, 3), (\texttt{control}, 1)\}$.

Counters and Counter Clauses

In many definitions there are state terms that would lead to fluent paths generating a large number of fluents (i.e. distinct subterms at the fluent path), each fluent having a numeric interpretation, for example being the count of play steps. We are eager to translate the part of the game dealing with such state terms compactly, using the real number facilities of Toss, since otherwise each “numeric value” would be considered structurally significant, leading to combinatorial explosion in the translation process.

In the future, the specification and implementation of counters handling, can be generalized for game definitions whose counters deviate from the (natural but very limited) format currently handled. Ideally, even “structural” functional dependencies should be recognized and handled compactly, but we leave it to future work.

Definition 13. A *numeric function relation* is a binary relation given in the game definition entirely by ground facts, containing as relation arguments only numerals: constants that can be parsed as (real) numbers, and the first arguments of its facts are distinct for distinct facts.

Example 14. In the game `pacman3p.gdl`, the numeric function relations are `++`, `-`, `succ`, `scoremap`. As it turns out, `++` and `-` are only used for structural purposes (which are not precluded by determining that a relation is a numeric function relation).

Definition 15. A `next` clause `(<= (next (`f h`)) `b`)`, where $f \in C$ is an unary functor and b is the clause body, is a *counter clause given counter candidates* C, when either h is a numeral (then it is a "counter reset clause"), or h is a variable, there is a directed path leading to it from some positive `true` literal `(true (`g h_0`))` in b with $g \in C$, using numeric function relation positive literals in b (for example an empty path when `(true (`g h`))` is in b), and the variables on the path (including h) do not have other occurrences in b; in other words, when h is computed from a counter candidate using numeric function relations and the computation does not participate in the remaining "structural" condition of the clause body.

We find the set of `counters` by iterating an operator counters(C), that finds functors f whose all `(<= (next (`f h`)) `b`)` clauses are conter clauses given counter candidates C. We start with all unary functors f that have exactly one `init` fact `(<= (init (`f h`)))`, and h is a numeral.

We remove the `next` clauses that build counters before the aggregate playout that generates all state terms is performed, and add them back after the fluent paths are found. Therefore, no fluent path will point into a counter.

We prepare the translation of numeric function relations for later use: they are translated as piecewise-linear functions.

Structure Elements

The fluent paths define the partition of GDL state terms into elements of the Toss structures in the following way.

Definition 16. We define the *element coordinate equivalence* $\sim$ by:

$$t \sim s \quad \Leftrightarrow \quad t[\mathcal{P}_f \leftarrow c] = s[\mathcal{P}_f \leftarrow c] \text{ for all terms } c.$$

The set of elements A of the initial Toss structure $\mathfrak{A}$ consists of equivalence classes of $\sim$. For $a \in A$ we write $[\![a]\!]$ to denote the corresponding subset of equivalent terms from $\mathcal{S}$.

We define *coordinate paths* $\mathcal{P}_c$ as such paths p that, for all $a \in A$, if, for any $t \in [\![a]\!]$, $t\downarrow_p$, then for all $s, t \in [\![a]\!]$, $s\downarrow_p = t\downarrow_p$. For $p \in \mathcal{P}_c$ we can therefore define the *coordinate subterm* $a\downarrow_p^c$ as $t\downarrow_p$ for $t \in [\![a]\!]$.

Example 17. Continuing the example of the Tic-tac-toe specification from Figure 5.1, we construct the set A. The terms in $\mathcal{S}$ are either $(\texttt{cell}\ s\ t\ p)$ or $(\texttt{control}\ q)$, where s and t range over `a, b, c`, p over `x, o, b` and q can be `x` or `o`. Since $\mathcal{P}_f = \{(\texttt{cell},3),(\texttt{control},1)\}$, we consider as $\sim$-equivalent all `cell` terms which differ only on p and all `control` terms which differ on q. Thus, the set A consists of 10 elements: the element a_{ctrl} for the single equivalence class of `control` terms, and 9 elements $a_{s,t}$ for the equivalence classes of $(\texttt{cell}\ s\ t\ p)$ with fixed s and t.

$$\begin{aligned} A = \{a_{ctrl},\ & a_{\mathtt{a},\mathtt{a}},\ a_{\mathtt{a},\mathtt{b}},\ a_{\mathtt{a},\mathtt{c}},\\ & a_{\mathtt{b},\mathtt{a}},\ a_{\mathtt{b},\mathtt{b}},\ a_{\mathtt{b},\mathtt{c}},\\ & a_{\mathtt{c},\mathtt{a}},\ a_{\mathtt{c},\mathtt{b}},\ a_{\mathtt{c},\mathtt{c}}\}. \end{aligned}$$

Note the similarity to the starting structure in Figure 1.2, up to the control element. The set of coordinate paths for this specification is $\mathcal{P}_c = \{(\texttt{cell},1),(\texttt{cell},2)\}$.

Optionally, we refine $\mathcal{P}_c$ afterwards to point to the leaves of the state terms that contain a path, among all ground state terms.

5.2.2 Expanding the GDL Game Definition

Now we discuss transformations of the game G that result in a longer (having more clauses) but simpler definition. *Eliminating a GDL variable $x \in FV(\mathcal{C})$ by a set of terms T* means replacing the clause C the variable occurs in, with a set of clauses $\mathcal{C}_t = \mathcal{C}[v \leftarrow t]$ for $t \in T$. First such transformation initiated the translation process: we eliminated variables ranging over players (by virtue of occurring in `does`, `legal` or `goal` atoms), by the players of the game.

Also at the beginning of the translation process, we removed counter building clauses from the game definition; we add them back at this point.

We transform the definition G by inlining all relations other than `next` whose defining clauses have occurrences of `does` atoms. It is required because later we need to reliably partition `next` clauses according to their `does` atoms.

Before generating Toss formulas we transform the definition G by grounding all variables that have occurrences at fluent paths, i.e. eliminating these variables by constants that occur at these paths in ground state terms $\mathcal{S}$.

We eliminate pattern matching when possible: arguments of a defined relation that are not variables in the head of each clause defining the relation. We generate a new relation name for each root functor of a value of the argument. We replace all atoms of the old relation by those new relations to which the eliminated argument of the atom can be instantiated, if necessary duplicating the clause containing the atom (if some variables need to be eliminated by the instantiation). If the eliminated argument is a variable but the eliminating functor has positive arity, we introduce fresh variables to instantiate the variable with.

As a special case of eliminating pattern matching, we eliminate arguments of a defined relation that are ground in the head of each clause defining the relation.

As an optimization, instead of duplicating the clause, if a variable is local to an atom (in all cases of eliminating a variable), we can replace the atom by a disjunction of corresponding atoms, or if it is a negative literal, by a conjunction of negated atoms.

For simplicity, we still refer to the transformed definition as G, but it is to be understood as the result of transformation G' equivalent to the original game definition G.

5.2.3 Relations

Preparing the Translation of Relations

Prior to translating formulas, we need to transform the game definition, iteratively for each relation in the partial order of the "call graph", i.e. whenever possible doing the transformation for a relation after doing it for relations used to define it. (We keep the convention from Section 5.2.2, that G is substituted with the transformed clauses.) We process both future structure relations and future defined relations. First we determine by which coordinate paths to pass arguments to the relation (and whether to collapse original arguments). Then we add missing state terms to the relation call sites, as each translated relation's argument needs a corresponding state term.

State Terms to Transfer Arguments We prepare relation R with GDL defining clauses

$$(\texttt{<= (R t}_1^1 \ldots \texttt{t}_n^1\texttt{) b}_1\texttt{)}, \ldots, (\texttt{<= (R t}_1^k \ldots \texttt{t}_n^k\texttt{) b}_k\texttt{)}$$

Let all atoms of R in G (including both the heads $(R\ t_1^j \ldots t_n^j)$, and inside of b_j above) be $\mathcal{R} = \left\{(R\ r_1^1 \ldots r_n^1), \ldots, (R\ r_1^K \ldots r_n^K)\right\}$. Based on $\mathcal{R}$ we will find a partition of argument positions and an assignment of coordinate paths to positions $\text{ArgPaths}(R) = \left((o_1, p_1), \ldots, (o_n, p_n)\right)$ such that $\{o_1, \ldots, o_n\} = \{1, 2, \ldots, N\}$, for any partition class $\mathcal{I}_I = \{i \mid o_i = I\}$, the paths $(p_i \mid i \in \mathcal{I})$ are distinct and do not conflict, i.e. $(\exists s)(\forall p_i \mid i \in \mathcal{I})\ s \downarrow_{p_i}$. (Equivalently, $\text{ArgPaths}(R) = \left\{\left\{(i, p_i) \mid i \in \mathcal{I}_1\right\}, \ldots, \left\{(i, p_i) \mid i \in \mathcal{I}_N\right\}\right\}$.)

GDL arguments of a single partition class will be passed as a single relation argument.

To find the paths and the partition, consider a clause body $\texttt{b}$, any occurrence of relation R atom $(R\ r_1^j \ldots r_n^j)$ in $\texttt{b}$ and positive literal $(\texttt{true}\ s) \in \texttt{b}$ (where the literal is not under disjunction). Let $\{p, i \mid s \downarrow_p = r_i^j\}$. We count such sets of paths for all $\texttt{b}$ and positive $(\texttt{true}\ s) \in \texttt{b}$. We greedily select sets that together cover all argument positions, with highest size, and of equally sized with highest count. Of these, we build

the partition by removing from the sets the path-position pairs where the position is already present in remaining path-position pairs, in order reverse to the selection criterion.

In case no set of paths contains a path for the ith argument, we set the path $p_i \in \mathcal{P}_c$ (with a unique o_i) so that the intersection of the projection of the graph of R for the initial game state $g_{R,i} = \{s | G \vdash R(t_1, \ldots, t_n)$ *for any* $\bar{t}$ *s.t.* $t_i = s\}$, and the set of subterms of state terms at path p_i, $g_{p_i} = \{s \downarrow_{p_i} | s \in \mathcal{S}\}$, is maximal w.r.t. cardinality ($p_i = \arg\max_{p \in \mathcal{P}_c} \left| g_{R,i} \cap g_p \right|$).

Ideally, $p_i \in \mathcal{P}_c$ should be a path whose domain, i.e. the set $\left\{t \middle| s \downarrow_{p_i} = t, s \in \mathcal{S}\right\}$, contains the domain of the ith argument of R, i.e. the sum of projections of R on ith argument for all possible game states. We do not guarantee this.

We single out relations that are static, with only singleton partition classes of arguments, and where there were multiple candidate paths for some argument. We call them *coordinating relations*, and for each coordinating relation R we remember all the combinations of candidate argument paths

$$\text{Coordin}(R) = \left\{(p_1, \ldots, p_n) \mid p_i \textit{ is an } i\textit{th argument path candidate}\right\}$$

Once the paths for arguments have been selected, we make sure that a clause in G that has an atom $(R\ r_1^1 \ldots r_n^1)$, has the positive literals $(\texttt{true}\ s_{\mathcal{I}})$ such that $\bigwedge_{i \in \mathcal{I}} s_{\mathcal{I}} \downarrow_{p_i} = r_i^j$. For every $\mathcal{I}$ for which such a positive literal does not occur in the clause body, we add an atom $\left(\texttt{true}\ \texttt{BL}(\{p_i \leftarrow r_i^j \mid i \in \mathcal{I}\})\right)$ to the clause. The notation $\texttt{BL}(\{p_i \leftarrow t_i \mid i \in \mathcal{I}\})$ for paths p_i and terms t_i denotes a state term containing t_i at path p_i, and `BLANK` as subterms at all its positions that are not on any path p_i (i.e. are not prefixes of any p_i). In case of a coordinating relation R, if for none of paths p_i : $(\ldots, p_i, \ldots) \in \text{Coordin}(R)$, such an $s \downarrow_{p_i} = r_i^j$ exists, we add $\left(\texttt{true}\ \texttt{BL}(p_i \leftarrow r_i^j)\right)$ for some such path p_i.

Relations in the Structure

Having defined the elements A as equivalence classes of state terms, let us now define the relations in the initial structure $\mathfrak{A}$.

Subterm equality relations. For all pairs of paths $p, q \in \mathcal{P}_c$ we introduce the *subterm equality relation* $Eq_{p,q}$:

$$Eq_{p,q}(a_1, a_2) \iff a_1 \downarrow_p^c = a_2 \downarrow_q^c .$$

Coordinating relations. For all coordinating relations R, and all tuples of paths $(p_1, \ldots, p_n) \in \text{Coordin}(R)$, we introduce the *coordinating relation* $R_{p_1,\ldots,p_n}$:

$$R_{p_1,\ldots,p_n}(a_1, \ldots, a_n) \iff R(a_1 \downarrow_{p_1}^c, \ldots, a_n \downarrow_{p_n}^c) \text{ in any state.}$$

Fact relations. For all remaining relations R of G that do not (directly or indirectly) depend on the state, let $\text{ArgPaths}(R) = \big((o_1, p_1), \ldots, (o_n, p_n)\big)$. Let $o^{-1}(j) = i$ iff $o_i = j$. We introduce the *fact relation* R:

$$R(a_1, \ldots, a_N) \iff R(a_{o^{-1}(1)} \downarrow^c_{p_1}, \ldots, a_{o^{-1}(n)} \downarrow^c_{p_n}) \text{ in any state.}$$

Coordinate predicates. For all paths $p \in \mathcal{P}_c$ and subterms $s = t \downarrow_p, t \in \mathcal{S}$, we introduce the *coordinate predicate* $Coord^s_p(a)$:

$$Coord^s_p(a) \iff a \downarrow^c_p = s.$$

Fluent predicates. Let $\mathcal{S}^{\text{init}} = \{s \mid \texttt{init}(s) \in G\}$ be the set of state terms under `init`. For all paths $p \in \mathcal{P}_f$ and subterms $s = t \downarrow_p, t \in \mathcal{S}$, we introduce the *fluent predicate* $Flu^s_p(a)$:

$$Flu^s_p(a) \iff t \downarrow_p = s \text{ for some } t \in [\![a]\!] \cap \mathcal{S}^{\text{init}}.$$

Currently in the implementation, the string representing the path p alone is used as the predicate name, we use the prefixes *Coord* and *Flu* in the reference for clarity.

Root predicates. We define the coordinate root relation $\sim_m$ by:

$$t \sim_m s \quad \Leftrightarrow \quad t[\mathcal{P}_f \cup \mathcal{P}_c \leftarrow c] = s[\mathcal{P}_f \cup \mathcal{P}_c \leftarrow c] \text{ for all terms } c.$$

We call an equivalence class of $\sim_m$ a *coordinate root*. For all coordinate roots m we introduce the *coordinate predicate* $Root_m$. Root predicates are similar to the coordinate predicates, but instead of matching against a subterm, they match against the coordinate root.

$$Root_m(a) \iff [\![a]\!] \subset m.$$

Example 18. To list the relations derived for the Tic-tac-toe specification, recall that $\mathcal{P}_c = \{(\texttt{cell}, 1), (\texttt{cell}, 2)\}$ consists of two paths. To shorten notation, we will just use the index i for $(\texttt{cell}, i)$.

Subterm equality relations. The relation $Eq_{i,j}$ contains all pairs of elements for which the ith coordinate of the first one equals the jth coordinate of the second one. For example

$$\begin{aligned} Eq_{1,1} = \{&(a_{\texttt{a,a}}, a_{\texttt{a,a}}), (a_{\texttt{a,a}}, a_{\texttt{a,b}}), (a_{\texttt{a,a}}, a_{\texttt{a,c}}), \\ &\ldots \\ &(a_{\texttt{c,c}}, a_{\texttt{c,a}}), (a_{\texttt{c,c}}, a_{\texttt{c,b}}), (a_{\texttt{c,c}}, a_{\texttt{c,c}})\} \end{aligned}$$

describes the identity of the first coordinate of two cells.

Coordinating relations. The only relation in the example specification is `nextcol` and we thus get the relations $\texttt{nextcol}_{i,j}$. For example, the relation

$$\begin{aligned}\texttt{nextcol}_{2,2} = \{&(a_{\texttt{a,a}}, a_{\texttt{a,b}}), (a_{\texttt{a,a}}, a_{\texttt{b,b}}), (a_{\texttt{a,a}}, a_{\texttt{c,b}}),\\ &\ldots,\\ &(a_{\texttt{c,b}}, a_{\texttt{a,c}}), (a_{\texttt{c,b}}, a_{\texttt{b,c}}), (a_{\texttt{c,b}}, a_{\texttt{c,c}})\}\end{aligned}$$

contains pairs in which the second element is in the successive row of the first one. Note that, for example, the formula $Eq_{1,1}(x_1, x_2) \wedge \texttt{nextcol}_{2,2}(x_1, x_2)$ specifies that x_2 is directly right of x_1 in the same row.

Coordinate predicates. Since the terms inside `cell` at positions 1 and 2 range over `a, b, c`, we get 6 coordinate predicates $Coord_i^{\texttt{a}}, Coord_i^{\texttt{b}}, Coord_i^{\texttt{c}}$ for $i = 1, 2$. They mark the corresponding terms, e.g.

$$Coord_2^{\texttt{a}} = \{a_{\texttt{a,a}}, a_{\texttt{b,a}}, a_{\texttt{c,a}}\}$$

describes the bottom row.

Fluent predicates. The fluent paths $\mathcal{P}_f = \{(\texttt{cell}, 3), (\texttt{control}, 1)\}$ and the terms appearing there are `b, x, o` for $(\texttt{cell}, 3)$ and `x, o` for $(\texttt{control}, 1)$, resulting in 5 fluent predicates. For example, $Flu^{\texttt{o}}_{(\texttt{cell},3)}(a)$ will hold exactly for the elements a which are marked by the player `o`. In the initial structure, the only nonempty fluent predicates are

$$Flu^{\texttt{b}}_{(\texttt{cell},3)} = A \setminus \{a_{ctrl}\} \quad \text{and} \quad Flu^{\texttt{x}}_{(\texttt{control},1)} = \{a_{ctrl}\}.$$

Root predicates. For the specification we consider, there are two coordinate roots: $m_1 = \left\{(\texttt{control } x) \;\middle|\; (\texttt{control } x) \in \mathcal{S}\right\}$ and $m_2 = \left\{(\texttt{cell } x\ y\ z) \;\middle|\; (\texttt{cell } x\ y\ z) \in \mathcal{S}\right\}$. The predicate $Root_{m_1} = \{a_{ctrl}\}$ holds exactly for the control element, and $Root_{m_2} = A \setminus \{a_{ctrl}\}$ contains these elements of A which are not the control element, i.e. the board elements.

In Toss, *stable relations* are relations that do not change in the course of the game, and *fluents* are relations that do change. Roughly speaking, a fluent occurs in the symmetric difference of the sides of a structure rewrite rule. In the translation, the fluent predicates Flu^s_p are the only introduced fluents, i.e. these predicates will change when players play the game and all other predicates will remain intact.

Counters in the Initial Structure If there are any counters in the definition, we add an element `COUNTER`. We add a singleton predicate `COUNTER` ranging only over the element `COUNTER`. For each counter, we introduce a corresponding function over `COUNTER`, with the initial value determined by an `init` fact from the game definition.

5.2.4 Structure Rewriting Rules

To create the structure rewriting rule for the Toss game, we first construct two types of clauses and then transform them into structure rewriting rules. Let $(p_1, \ldots, p_n)$ be the players in G, i.e. let there be `(role` p_1`)` up to `(role` p_n`)` facts in G, in this order.

Move Clauses

By GDL specification, a legal joint move of the players is a tuple of player term – move term pairs which satisfy the `legal` relation. For a joint move $(m_1, \ldots, m_n)$ to be allowed, it is necessary that there is a tuple of `legal` clauses $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, with head of $\mathcal{C}_i$ being `(legal` p_i l_i`)`, and the `legal` arguments tuple being more general than the joint move tuple, i.e. $m_i \leq l_i$ for each $i = 1, \ldots, n$.

The move transition is computed from the `next` clauses whose all `does` relations are matched by respective joint move tuple elements as follows.

Definition 19. Let $\mathcal{N}$ be a `next` clause. The $\mathcal{N}$ *does facts*, $d_1(\mathcal{N}), \ldots, d_n(\mathcal{N})$, are terms, one for each player, constructed from $\mathcal{N}$ in the following way. Let `(does` p_i d_i^j`)` be all `does` facts in $\mathcal{N}$.

- If there is exactly one d_i for player p_i we set $d_i(\mathcal{N}) = d_i$.
- If there is no `does` fact for player p_i in $\mathcal{N}$ we set $d_i(\mathcal{N})$ to a fresh variable.
- If there are multiple $d_i^1, \ldots, d_i^k$ for player p_i we compute $\sigma = \mathrm{MGU}(d_i^1, \ldots, d_i^k)$ and set $d_i(\mathcal{N}) = \sigma(d_i^1)$.

We have $m_i \leq d_i(\mathcal{N})$ for each `next` clause $\mathcal{N}$ contributing to the move transition, since otherwise the body of $\mathcal{N}$ would not match the state enhanced with `(does` p_i m_i`)` facts.

Example 20. In the Tic-tac-toe example, there are three clauses where the control player is `o`, which after renaming of variables look as follows.

```
N1 = (<= (next (control x)) (does x noop)),
N2 = (<= (next (cell ?x2 ?y2 o))
         (does o (mark ?x2 ?y2))),
N3 = (<= (next (cell ?x3 ?y3 ?c))
         (true (cell ?x3 ?y3 ?c))
         (does o (mark ?x1 ?y1))
         (or (distinct ?x3 ?x1) (distinct ?y3 ?y1))).
```

The does facts are, respectively,

$$d_1(\mathcal{N}_1) = \texttt{noop} \quad \text{and} \quad d_2(\mathcal{N}_1) = x_{f_1},$$
$$d_1(\mathcal{N}_2) = x_{f_2} \quad \text{and} \quad d_2(\mathcal{N}_2) = (\texttt{mark}\ x_2\ y_2),$$
$$d_1(\mathcal{N}_3) = x_{f_3} \quad \text{and} \quad d_2(\mathcal{N}_3) = (\texttt{mark}\ x_1\ y_1).$$

Each rewrite rule of the translated game is generated from a tuple of `legal` clauses $\mathcal{C}_1, \ldots, \mathcal{C}_n$ and a selection of `next` clauses $\mathcal{N}_1, \ldots, \mathcal{N}_m$, with variables renamed so that no variable occurs in multiple clauses, and such that

$$l_i \doteq d_i(\mathcal{N}_1) \doteq \ldots \doteq d_i(\mathcal{N}_m)$$

for each player p_i. We will consider all tuples $\overline{\mathcal{C}}, \overline{\mathcal{N}}$ for which the the above MGU exists and we will denote it by $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$. We apply $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ to the clauses and we will refer to the result simply as *the `legal` and `next` clauses of the rule*.

Technically, for completeness, we need to generate a rule for a set of `next` clauses even if we generate a rule for its superset, and then for correctness, we need to preclude application of the first (more general) rule when the more concrete rule is applicable, adding `distinct` conditions to clauses of the otherwise more general rule. In the current implementation, we select a minimal covering family of maximal sets of `next` clauses, where covering means that every clause occurs in at least one set of the family. (While in Section 5.2.4 we describe additional partition of the substituted clauses, in unlikely scenarios the generated $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ might be too specific to capture all possible moves.)

Example 21. Let $\mathcal{C}_1 = \texttt{noop}$ and $\mathcal{C}_2 = (\texttt{mark}\ x\ y)$. The clauses $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ introduced above form a maximal set,

$$\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}} = \{x_{f_1} \mapsto (\texttt{mark}\ x\ y), \quad x_{f_2} \mapsto \texttt{noop},$$
$$x_2 \mapsto x, \quad y_2 \mapsto y, \quad x_1 \mapsto x, \quad y_1 \mapsto y\}.$$

With all tuples $\overline{\mathcal{C}}, \overline{\mathcal{N}}$ selected and the MGU $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ computed, we are almost ready to construct the rewriting rules. Still, for a fixed tuple $\overline{\mathcal{C}}, \overline{\mathcal{N}}$, we first need to compute "erasure clauses" to make sure that a rewrite rule captures the whole substructure that should be affected.

Concurrent Moves Games Introduced in Section 5.2.7, concurrent moves games use a factored approach: since the d_i never share variables, `legal` and `next` clauses are assigned to players and the whole construction of structure rewriting rules is done separately for each player. Clauses without a `does` atom are assigned to the "environment". (In the interpretation, to reuse code, we simply build single-term `legal` tuples for concurrent moves games.)

Erasure Clauses

So far, we have not accounted for the fact that rewrite rules of Toss only affect the matched part of the structure, while the GDL game definition explicitly describes the construction of the whole successive structure. Negating the frame clauses from the tuple $\overline{\mathcal{N}}$ and transforming them into *erasure clauses* will keep track of the elements that possibly lose fluents and ensure correct translation.

We determine which clauses are frame clauses prior to partitioning into the rule clauses and computing the substitution $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ – at the point where fluent paths are computed. We transform frame clauses by expanding relations that would otherwise be translated as defined relations: so as to eliminate local variables whenever possible.

From the frame clauses in $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_1),\ldots,\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_m)$, we select subsets J such that, clauses in J having the form (<= (`next` $\mathtt{s_i}$) $\mathtt{b_i}$), it holds

$$s_1 \doteq_f \ \ldots \ \doteq_f s_{|J|},$$

i.e. the arguments of `next` unify. Note that we use $\doteq_f$ instead of the standard unification, and by that we mean that the variables shared with the `legal` clauses $\overline{\mathcal{C}}$ are treated as constants. The reason is that these variables are not local to the clauses and must therefore remain intact. As before, we select a minimal covering family of maximal such subsets (possibly resulting, in unlikely cases, in rules that do not remove fluent predicates over elements that do not gain fluent predicates during rewriting.)

Intuitively, the selected sets J describe a partition of the state terms that could possibly be copied without change by the rule we will generate for $\overline{\mathcal{C}},\overline{\mathcal{N}}$.

Let us write ρ for the f-MGU of $s_1,\ldots,s_{|J|}$. To compute the bodies of the erasure clauses, we negate the disjunction of substituted bodies of the frame clauses and bring this Boolean combination to disjunctive normal form (DNF), i.e. we compute conjunctions $e_1,\ldots,e_l$ such that

$$\neg(\rho(b_1)\vee\cdots\vee\rho(b_{|J|})) \ \equiv \ (e_1\vee e_2\ldots\vee e_l).$$

As the head of each erasure clause we use $\rho(s_1)=\cdots=\rho(s_{|J|})$.

Erasure clauses that contain variables other than the "fixed variables" of the `legal` clauses are problematic. If such "unfixed" variables appear in the head, the erasure clause should apply to all instantiating variables in a single application of the rewrite rule, which is too difficult to achieve using Toss semantics. Rather, we match $\rho(s_1)$ against positive atoms in the bodies of the move clauses and add an instantiated clause for each match. If such "unfixed" variables occur only in the body, they should be quantified universally. We just ignore erasure clauses containing unfixed variables that occur only in the body.

The resulting erasure clauses are

$$\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(J) = \{(\texttt{<=}\ \sigma(\rho(s_1))\ \sigma(e_i)) \mid \sigma \textit{ unifies } \rho(s_1) \textit{ with a pos. atom of } \overline{\mathcal{N}} \vee (\sigma = id \wedge FVar((\texttt{<=}\ \texttt{h}\ \texttt{e}_\texttt{i})) \setminus FVar(\overline{\mathcal{C}}) = \varnothing), i = 1, \ldots, l\}$$

and we write $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ for the union of all $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(J)$, i.e. for the set of all $\overline{\mathcal{C}},\overline{\mathcal{N}}$ erasure clauses.

Example 22. In our example, $\mathcal{N}_3$ and its counterpart for the other player are the only frame clauses in G. After negation, $\sigma(\mathcal{N}_3)$ splits into several clauses e_i. The relevant one is `(<= (next (cell ?x3 ?y3 ?c)) (?x3 = ?x) (?y3 = ?y))`, i.e. `(<= (next (cell ?x ?y ?c)))`. The resulting erasure clause is `(<= (next (cell ?x ?y BLANK)))`. If no other clause had the form `(<= (next (cell ?x ?y ...)) ...)`, this clause would cause the erasure of any fluent at coordinates (x, y). Other erasure clauses derived from $\sigma(\mathcal{N}_3)$ turn out to be contradictory with remaining clauses, and thus will not contribute to any rewrite rule in the translation, due to filtering described below.

Unframed Fluents Some possible subterms of state terms are not covered by any frame clauses. We cannot rely then on erasure clauses to guarantee that matching those fluents will be part of the $\mathfrak{L}$-structure of the rewrite rule, rather than the precondition. For all such unframed fluents that occur in the $\mathfrak{R}$-structure of the rewrite rule, we move their atoms from the precondition to the $\mathfrak{L}$-structure. But note that different variables existentially quantified in a precondition need not have distinct assignments, thus we cannot require that elements added to "erase" unframed fluents have distinct assignments when matching a rule. To bypass this problem, we extend the semantics of rewrite rules, providing a new special relation `nondistinct`, that selectively weakens the embedding condition of rule matching, into a homomorphism condition where only the elements not related by `nondistinct` need to be distinct.

Example 23. In the Pacman game `pacman3p.gdl`, the pacman and the ghosts are singleton predicates, and therefore do not have corresponding frame (and erasure) clauses. We therefore move the check for their old location from the precondition of a rule to the $\mathfrak{L}$-structure of a rule, to erase them from the old location before adding to the new location. But we cannot rely that the old location and the new location are distinct. For example, the definition of `(move nowhere)` action is provided just as a move in any (other) direction, also describing the old location and the new location. We join the old and the new location by `nondistinct` in the $\mathfrak{L}$-structure.

Rewriting Rule Creation

For each suitable tuple $\overline{\mathcal{C}},\overline{\mathcal{N}}$ we have now created the unifier $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ and computed the erasure clauses $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$. To create the rules, we need to further partition the *rule clauses* $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{C}_i), \sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_i)$ and $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$, and augment them with further conditions. The reason is that the prepared rule clauses may have different matches in different game states, while the Toss rule has to be built from all the rule clauses that would match when the Toss rule matches. Therefore, we need to build a Toss rule for each subset of rule clauses that are "selected" by some game state (i.e. are exactly the rule clauses matching in that state), but add to it "separation conditions" that prevent the Toss rule from matching in game states where more rule clauses can match.

We select groups of atoms (collected from rule clauses) that separate rule clauses, and generate a Toss rule candidate for every partition of the groups into true and false ones: we collect the rule clauses that agree with the given partition. The selected atoms, some negated according to the partition, form the separation condition. Currently, we do not consider atoms under disjunction (mostly for simplicity considerations; would this cause problems, the definition can be extended to include disjunctions in making the partition). We remove from the separation condition negations of atoms that contain "local variables": variables not appearing in positive atoms of the whole condition.

We filter the rule candidates by checking for satisfiability (in the same GDL model as used for building the initial Toss structure) of the static part of their clause bodies, and later by checking for satisfiability of the whole clause bodies in at least one of a collection of random playout states. For each remaining candidate, we will construct the Toss rule in two steps.

In the first step we generate the *matching condition*: we translate the conjunction of the bodies of rule clauses and the separation condition. This translation follows the definitions of atomic relations presented in Section 5.2.3 and is described in Section 5.2.6.

In the second step, we build a Toss rewrite rule itself. From the heads of the move clauses of a rule candidate, we build the `ADD` part of the rule effect; from heads of the erasure clauses we build the `DELETE` part. We add "blanked-out" heads of the clauses (atoms $(\texttt{true}\ \texttt{BL}(s))$ for clause heads $(\texttt{next}\ s)$) to the matching condition, not to lose any facts constraining the rule structure elements (but only after the rule candidates are checked for satisfiability). The precondition of the Toss rule is built from the matching condition. Quantification in the precondition over variables occurring in the `ADD` and `DELETE` parts is dropped.

Translating Counter Clauses We treat the counter clauses specially, to simplify presentation we describe the differences here. We do not add the "blanked-out" counter state term to the $\mathfrak{R}$-structure, instead we add a new element (with the

COUNTER predicate over it in the $\mathfrak{L}$-structure). We remove the computation of the counter update from the counter clause, reset the resulting clause to "unrequired", and add it back to the candidate clauses, remembering the association with the counter update. After the rule clauses have been finally partitioned, we make sure that there is only one counter clause for a given counter in each resulting rule candidate. To complete the translation, we calculate the function update, inlining the translations of numeric function relations, and add the update to the rule translation for each counter occurring in a rule.

Having constructed and filtered the rewriting rule candidates, we have almost completed the definition of $T(G)$. Payoff formulas are derived by instantiating variables standing for the `goal` values. The formulas defining the `terminal` condition and specific `goal` value conditions are translated as described in Section 5.2.6, from disjunctions of bodies of their respective clauses.

5.2.5 Translating Moves between Toss and GDL

To play as a GDL client, we need to translate legal moves from G into Toss rule embeddings for $T(G)$, and conversely, the rule embeddings from $T(G)$ into moves of G.

In the incoming move case, we augment the Toss rewrite rules with constraints provided in the incoming move, try to embed each of the augmented rules, and return the single rule that matches and its unique embedding. Augmenting the rule is done in the following simple way: If the head of a `legal` clause of the rule contains a variable v at path q, a Toss variable x was derived from a game state term t such that $t \downarrow_p = v$, and the incoming move has term s at path q, then we add $Coord_p^s(x)$ to the precondition.

It is actually possible, that there is more than a single rule and/or multiple embeddings; the semantics of the original GDL game specification demands that all rules should be applied over all their non-overlapping embeddings. For the most part we do not worry about such exceptional situations, but we handle one natural case in Paragraph 5.2.7.

To translate the outgoing move, we recall the heads of the `legal` clauses of the rule that is selected, as we only need to substitute all their variables. To eliminate a variable v contained in the head of a `legal` clause of the rule, we look at the rule embedding; if $x \mapsto a$, x was derived from a game state term t such that $t \downarrow_p = v$, and $a \downarrow_p^c = s$, then we substitute v by s. The move translation function μ is thus constructed.

5.2.6 Translating Formulas and Building Defined Relations

We translate a GDL relation as either multiple Toss stable relations (i.e. structure relations that do not change during the game), or as Toss defined relation (i.e. a re-

lation given by its defining formula). All GDL relations that even indirectly depend on `true` need to be translated as defined relations. Of the remaining relations, we select the ones to be translated as structure (stable) relations heuristically. Currently, a parameter of the translation allows to: select relations with arity smaller than three; or, select relations whose (some, or all) defining clauses are ground (i.e. with empty bodies).

Translating Formulas

We normalize the GDL formula to be translated Φ, which is composed of conjunctions, disjunctions and literals, into a disjunction $\text{TrDistr}(\Phi) := \Phi_1 \vee \ldots \vee \Phi_n$, so that every $\Phi_i = G_i \wedge ST_i^+ \wedge ST_i^-$, where all literals in ST_i^+ are positive `true` atoms and all literals in ST_i^- are negated `true` atoms, excluding application over counter terms (we avoid unnecessary expansions), note that `true` atoms over counter terms are left in G_i. Let $\texttt{ST}(\varphi)$ be all the state terms, i.e. arguments of `true` atoms, in φ, that are not counter terms, and let $\texttt{CT}(\varphi)$ be the counter term atoms respectively.

For the purpose of translation involving counters, let C_i^V be an assignment of counters (identified by their names) to GDL variables $x \leftarrow c$ such that a positive atom $(\texttt{true}\ (c\ x))$ occurs in Φ_i. Also, let $CQ(\Psi)$ be $\exists v_C\big(\text{COUNTER}(v_C) \wedge \Psi\big)$ when $v_C \in \texttt{FreeVar}(\Psi)$, and $CQ(\Psi) = \Psi$ otherwise, where v_C is a distinguished variable for handling counters translation.

$\text{TrRels}(\varphi, S_1, S_2)$ descends φ translating each literal not involving counters as a conjunction of literals, for every combination of coordinate paths into S_1 state terms, such that at least one of those terms is from S_2.

When TrRels encounters a `true` atom over counter c, it builds an equation between $c(v_C)$ and the argument of c in the atom when it is a constant, or in case c is applied to a variable x, the value $c'(v_C)$ for $c' = C_i^V(x)$. If TrRels encounters a numeric function applied to either constants or variables in the domain of C_i^V, it applies the function to the right argument when building a similar equation. The translation will currently fail if relations other than numeric functions are applied to variables that also occur in counter terms: general case left as future work.

$\text{TrST}(\varphi)$ translates `true` atoms which are not counters as a conjunction of their coordinate and fluent predicates.

Let eqs_i be $\bigwedge\big\{\texttt{EQ}(x,x) | x \in \texttt{FreeVar}(\Phi_i) \setminus \texttt{FreeVar}(\texttt{CT}(\Phi_i))\big\}$. The relation name `EQ` serves technical purposes: we treat is as a coordinating relation, relations $\texttt{EQ}_{p,q}$ are identified with subterm equality relations $Eq_{p,q}$.

The result of translation is the disjunction of translations of each Φ_i. Let $\texttt{BL}(t) = t\big[\mathcal{P}_f \leftarrow \texttt{BLANK}\big]$. A single $\Phi_i = G_i \wedge ST_i^+ \wedge ST_i^-$ is translated as:

$$\begin{aligned}
\mathrm{Tr}(\Phi_i) \coloneqq \exists V^+\big(&\mathrm{TrRels}(eqs_i \wedge G_i, \mathtt{ST}(ST_i^+), V^+) \wedge \mathrm{TrST}(ST_i^+) \wedge \\
&\neg\exists V^-\big(\mathrm{TrRels}(eqs_i \wedge G_i, \mathtt{ST}(ST_i^+) \cup \mathtt{ST}(ST_i^-), V^-) \wedge \\
&\quad \mathrm{TrST}(\mathtt{NNF}(\neg ST_i^-))\big)\big) \\
V^+ \coloneqq &\big(\mathtt{BL}(\mathtt{ST}(ST_i^+)) \\
V^- \coloneqq &\big(\mathtt{BL}(\mathtt{ST}(ST_i^-)) \setminus V^+\big)
\end{aligned}$$

The result of translation is $\mathrm{Tr}(\Phi) \coloneqq CQ\big(\mathrm{Tr}(\Phi_1) \vee \ldots \vee \mathrm{Tr}(\Phi_n)\big)$. Note how variables with both positive and negative instantiating state terms are excluded from universal treatment; in particular, the variables corresponding to Toss rewrite rule structure elements will not be quantified universally, thanks to adding their "blank representants" to the rule condition.

We now proceed to define TrRels and TrST. For an atom r, let $\pm r$ mean either r or $\neg r$ when on the left-hand-side, and same-signed literal r or $\neg r$ on the right-hand-side as the one on the left-hand-side of an equality.

$$\begin{aligned}
\mathrm{TrRels}(\varphi_1 \wedge \varphi_2, S_1, S_2) =& \mathrm{TrRels}(\varphi_1, S_1, S_2) \wedge \mathrm{TrRels}(\varphi_2, S_1, S_2) \\
\mathrm{TrRels}(\varphi_1 \vee \varphi_2, S_1, S_2) =& \mathrm{TrRels}(\varphi_1, S_1, S_2) \vee \mathrm{TrRels}(\varphi_2, S_1, S_2) \\
\mathrm{TrRels}(\pm R(t_1, \ldots, t_n), S_1, S_2) =& \bigwedge \Big\{ \pm R_{p_1,\ldots,p_n}(v_1, \ldots, v_n) \;\Big|\; s_1, \ldots, s_n \in S_1 \wedge \\
& \{\mathtt{BL}(s_1), \ldots, \mathtt{BL}(s_n)\} \cap S_2 \neq \varnothing \wedge \\
& v_1 = \mathtt{BL}(s_1) \wedge \ldots \wedge v_n = \mathtt{BL}(s_n) \wedge \\
& p_1, \ldots, p_n \in \mathcal{P}_c \wedge s_1 \downarrow_{p_1} = t_1 \wedge \ldots \wedge s_n \downarrow_{p_n} = t_n \Big\} \\
& \textit{(when } R \textit{ is a coordinating relation)} \\
\mathrm{TrRels}(\pm \mathtt{true}(c(t))) =& \pm \big(c(v_C) - \mathtt{TrC}(t) = 0\big) \\
\mathrm{TrRels}(\pm R(t_1, t_2)) =& \pm \big(\big(\mathtt{NumF}(R)\big)(\mathtt{TrC}(t_1)) - \mathtt{TrC}(t_2) = 0\big) \\
& \textit{(when } R \textit{ can be translated as a numeric function } \mathtt{NumF}(R) \\
& \textit{and both } t_1 \textit{ and } t_2 \textit{ are in the domain of } \mathtt{TrC}\textit{)} \\
\mathtt{TrC}(n) =& n \quad \textit{(when } n \textit{ is a constant)} \\
\mathtt{TrC}(x) =& \big(C_i^V(x)\big)(v_C) \quad \textit{(when } x \textit{ is a variable in the domain of } C_i^V\textit{)} \\
\mathrm{TrST}(\varphi_1 \wedge \varphi_2) =& \mathrm{TrST}(\varphi_1) \wedge \mathrm{TrST}(\varphi_2) \\
\mathrm{TrST}(\varphi_1 \vee \varphi_2) =& \mathrm{TrST}(\varphi_1) \vee \mathrm{TrST}(\varphi_2) \\
\mathrm{TrST}(\mathtt{true}(t)) =& \bigwedge \big\{ Coord_p^s(v) \;\big|\; v = \mathtt{BL}(t) \wedge p \in \mathcal{P}_c \wedge t \downarrow_p = s \wedge s \neq \mathtt{BLANK} \big\} \wedge \\
& \bigwedge \big\{ Flu_p^s(v) \;\big|\; v = \mathtt{BL}(t) \wedge p \in \mathcal{P}_f \wedge t \downarrow_p = s \wedge s \neq \mathtt{BLANK} \big\} \wedge \\
& \bigwedge \big\{ Root_m(v) \;\big|\; v = \mathtt{BL}(t) \wedge t \in m \big\}
\end{aligned}$$

It remains to define TrRels for the case of fact relations. Let N, $p_1, \ldots, p_n$, $\mathcal{I}_1, \ldots, \mathcal{I}_N$

be as introduced for R. Since $(R\ r_1 \ldots r_n) \in \mathcal{R}$, there exist $s_1, \ldots, s_N \in \mathcal{S}_1$ such that $(\forall p_i \mid i \in \mathcal{I}_m)\ s_m \downarrow_{p_i}$. Put

$$\mathrm{TrRels}(\pm R(r_1, \ldots, r_n), S_1, S_2) = \pm R(\mathtt{BL}(s_1), \ldots, \mathtt{BL}(s_N)).$$

Translating Relation Definitions

Recall the notation introduced to generate state terms to transfer arguments for translating a relation R as a fact relation. Let $\mathrm{Coordin} = \big((a_1, p_1), \ldots, (a_n, p_n)\big)$ be the partition of R arguments and their paths, $\max_i a_i = N$, and $\mathcal{I}_m = \{i \mid a_i = m\}$ (for $m \in \{1, \ldots, N\}$). Let $v_1, \ldots, v_N$ be fresh Toss variables.

Recall that each $(R\ t_1^l \ldots t_n^l)$ is also a $(R\ r_1^{j_l} \ldots r_n^{j_l}) \in \mathcal{R}$ for some j_l. Therefore, there exist positive `true` literals $(\mathtt{true}\ s_1^l), \ldots, (\mathtt{true}\ s_N^l)$ in the body b_l such that $(\forall p_i \mid i \in \mathcal{I}_m)\ s_m^l \downarrow_{p_i}$. Let $V^l = \{\mathtt{BL}(s_1^l), \ldots, \mathtt{BL}(s_N^l)\}$. The translated definition of R is:

$$\begin{aligned} R(v_1, \ldots, v_N) =& \mathrm{TrDefR}((\mathtt{<=}\ (R\ t_1^1 \ldots t_n^1)\ b_1)) \\ & \vee \ldots \vee \\ & \mathrm{TrDefR}((\mathtt{<=}\ (R\ t_1^k \ldots t_n^k)\ b_k)) \\ \mathrm{TrDefR}((\mathtt{<=}\ (R\ t_1^l \ldots t_n^l)\ b)) =& (\exists V^l)\big(v_1 = \mathtt{BL}(s_1^l) \wedge \ldots \wedge v_N = \mathtt{BL}(s_N^l) \wedge \mathtt{Erase}_{V^l}(\mathtt{Tr}(b_l))\big) \end{aligned}$$

where $\mathtt{Erase}_V(\varphi)$ erases all quantification over variables from V in formula φ.

5.2.7 Concurrent Moves and Toss Locations

In Section 5.2.4, we described the creation of state-transition rewrite rules, i.e. where a single Toss rule is responsible for the transition to the next game state. But we also remarked that the rule clauses can be divided by players to whose actions they refer (by the `does` relation). In this case the game state transition is jointly described by several Toss rules that apply independently, each rule "enacted" by a player; such is the default way of defining simultaneous moves in Toss. We now elaborate on three modes of building the game graph in the translated game.

Turn-based Games

are games where in any game state there is at most a single player having genuine choice. Rather than attempting a complex analysis to detect as many turn-based games as possible, we recognize some cases where in all states, all players but one have a single legal move that is a constant (term of size one). Such move is conventionally called `noop`. In the current implementation we simply check what moves are available to players in the states of a couple of random playouts, so the

detection is unsound. We only handle the case where the alternation of control forms a cycle (players do not need to strictly alternate, for example a single-player game is also a turn-based game, as another example in a three-player game the first player may intersperse the moves of second and third player). We build a corresponding cyclic graph of Toss locations. We limit the turn-based translation to the case where all rule clauses have exactly one `does` atom (i.e. can be attributed to exactly one of the players).

Concurrent Moves Games

When translation as a turn-based game fails, but all rule clauses have at most one atom of `does` relation, we divide the clauses among players as mentioned earlier. We translate using a single-location game graph. The `next` clauses wihout `does` atoms, are assigned to a special player, "environment".

To synchronize the play, we introduce an additional element, or use an existing element singled-out by a stable singleton predicate, and we introduce *player marker* predicates $Player_{p_i}$ for each player p_i. We require that the player marker be absent in the corresponding player's rule left-hand-sides, and we add it over the singled-out element by each rule. Finally, we build a rule for the "environment" player that expects player markers for all players, and clears them all at once.

The `next` clauses assigned to the "environment" player together with the player marker clearing can form a single rewrite rule, or these can be separate rewrite rules.

General Interaction Games

When some rule clause has multiple `does` relations, we cannot use straightforward translation of the previous section. Instead, we use the "environment" player to carry out state changes, and have the players declare their moves by their rewrite rules.

This mode of translation differs from the "standard modes" in the way the `does` atoms are expanded by the `legal` clauses. For each `legal` clause we introduce a fresh GDL relation over the variables of the head of the `legal` clause. Instead of fully substituting `does` atoms by `legal` clause bodies in a `next` clause body, we now substitute by applications of the freshly introduced relations. Later, we translate the introduced relations coupled with `legal` clause bodies as defined relations. But we will not add the "`legal` defined relations" to the Toss game definition, instead we will use their defining formulas to derive rewrite rules for players. The introduced "`legal` defined relation names" will be the fluents added by the player rules, we will call them *move markers*. Unlike the game state fluents Flu_p^s, the move markers will usually have arity higher than one. Due to the injective nature of rewrite rules (the matchings are graph embeddings), we generate additional rewrite rules where

some arguments of move markers are made equal and the right-hand-sides of the rules are correspondingly smaller.

Besides move markers, player rules also introduce player markers, as in the previous section. The move and player markers are erased by the game state transition rule of the “environment” player.

5.2.8 Translation-specific Simplification

In Section 5.3, we describe general game simplification in Toss. Here, we describe parts of the translation process that were added to make the translation result more compact and efficient.

Introducing disequalities. When a defined relation is built from clauses that contain a single `distinct` positive literal each, when the defined relation is translated as a binary relation, a disjunction of those `distinct` literals would be semantically equivalent to a disequality of the relation arguments, and the remaining literals are the same for all bodies (modulo renaming of variables), the disequality is introduced to the translation and the `distinct` literals are removed.

Removing parts of `next` clauses that are redundant wrt. `legal` clauses. We remove literals in bodies of `next` clauses that are considered subsumed by literals from `legal` clause bodies. For a given `next` clause, we find the most coarse partition of its body such that different classes of this partition do not share variables. A group of the body literals is considered subsumed if its every literal unifies with some literal of some `legal` clause, under a common substitution. The current algorithm is greedy in that we only consider the first matching `legal` body literal for a given `next` body literal. We also ignore disjunctions.

Optionally, rather than considering literals sharing variables in bulk, we allow for single literals be removed as subsumed by a `legal` clause literal. The unifying substitution (i.e. the substitution that unifies the `next` clause literal and a `legal` clause literal) is applied to the `next` clause. Note that the literals may be intended for different elements as witnessed by remaining "next" clause literals, therefore this more aggresive pruning does not preserve correctness.

5.3 Game Simplification in Toss

Games automatically translated from GDL, as described above, are verbose compared to games defined manually for Toss. They are also inefficient, since the current solver in Toss works fast only for sparse relations.

Both problems are remedied by joining co-occurring relations. Relations which always occur together in a conjunction are replaced by their join when they are

over the same tuple. Analogically, we eliminate pairs of atoms when the arguments of one relation are reversed arguments of the other.

In an additional simplification, we remove an atom of a stable relation which is included in, or which includes, another relation, when an atom of the other relation already states a stronger fact. For example, if $Positive \subseteq Number$, then $Positive(x) \wedge Number(x)$ simplifies to $Positive(x)$, and $Positive(x) \vee Number(x)$ simplifies to $Number(x)$.

The above simplifications can be applied to any Toss definition. We perform one more simplification targeted specifically at translated games: We eliminate $Eq_{p,q}(x,y)$ atoms when we detect that $Eq_{p,q}$-equivalence of x and y can be deduced from the remaining parts of the formula.

The described simplifications are stated in terms of manipulating formulas; besides formulas, we also apply analogous simplifications to the structures of the Toss game: the initial game state structure, and the $\mathfrak{L}$ and $\mathfrak{R}$ structures of the rules.

6 Implementation

Toss consists of the main *TossServer* which is built from several main modules (components) explained in the sections below and corresponding to directories in the code tree. The main modules contain OCaml modules themselves. Figure 6.1 shows conceptual dependencies between components. The transitive closure of this dependency structure limits the dependencies between the underlying OCaml modules.

Toss includes a JavaScript client which provide the user interface.

Formula

This most basic directory implements formulas as described above and various operations on formulas which are necessary for other modules. It also contains a parser for formulas and the lexing file used for all parsers. The MiniSAT solver is included in this directory as it is used for formula simplification.

Solver

This directory contains the module which represents relational structures, and the full Solver, including the elimination-based solver for the theory of reals and the SAT-based solving algorithm for monadic second-order logic.

Arena

This directory contains modules which implement the game definition, including discrete and continuous structure rewriting, game file parser and client-server communication parser and request type.

Play

This directory contains modules responsible for automatic play, including the heuristic generation module, the abstract game tree module and its instantiations to Maximax and UCT.

GGP

This directory contains the code which translates GDL files into Toss format together with various needed simplifications. Multiple tests and a Java GGP Server are also included there to facilitate testing of the Toss-GGP code.

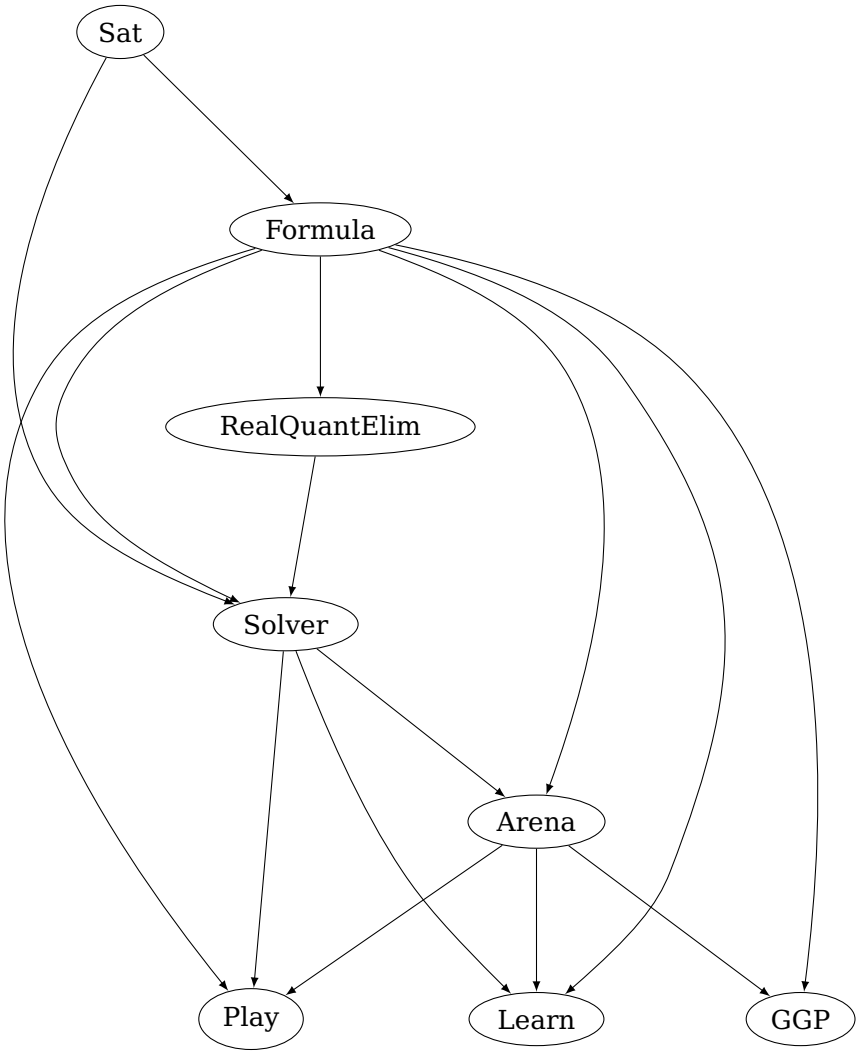


Figure 6.1: Dependencies among Toss components.

Server

In this directory we simply keep the implementation of TossServer together with several high-level tests to check that it works ok.

Bibliography

[1] Sven Burmester, Holger Giese, Eckehard Münch, Oliver Oberschelp, Florian Klein, and Peter Scheideler. Tool support for the design of self-optimizing mechatronic multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 10(3):207–222, 6 2008.

[2] John Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, October 2007.

[4] Richard O. Duda and Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[5] Kousha Etessami and Neil Immerman. Tree canonization and transitive closure. *Inf. Comput.*, 157(1-2):2–24, 2000.

[6] Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *SIAM-AMS Proceedings*, volume 7, pages 27–41, 1974.

[7] H. Gaifman. On local and non-local properties. In *Proc. of the Herbrand Symposium, Logic Colloquium'81*, pages 105–135. North Holland, 1982.

[8] T. Ganzow and Ł. Kaiser. New algorithm for weak monadic second-order logic on inductive structures. In *Proc. of CSL'10*, LNCS. Springer, 2010.

[9] Sylvain Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. Dissertation, University of Paris Sud, 2007.

[10] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[11] Erich Grädel. Why are modal logics so robustly decidable? *Bulletin of the European Association for Theoretical Computer Science*, 68:90–103, 1999.

[12] Erich Grädel, Ph. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Y. Vardi, Y. Venema, and S.Weinstein. *Finite Model Theory and Its Applications*. Texts in Theoretical Computer Science. Springer, 2007.

[13] Martin Grohe. Fixed-point logics on planar graphs. In *Proc. of LICS'98*, pages 6–15, 1998.

[14] Martin Grohe. Equivalence in finite-variable logics is complete for polynomial time. *Combinatorica*, 19(4):507–532, 1999.

[15] Martin Grohe. Fixed-point definability and polynomial time on graphs with excluded minors. In *Proc. of LICS'10*, pages 179–188, 2010.

[16] Joseph Y. Halpern and Rafeal Pass. Iterated regret minimization: A new solution concept. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 153–158, 2009.

[17] W. Hanf. Model-theoretic methods in the study of elementary logic. In J. Addison, L. Henkin, and A. Tarski, editors, *The Theory of Models*, pages 132–145. North Holland, 1965.

[18] Neil Immerman. Relational queries computable in polynomial time. In *Proc. of STOC'82*, pages 147–152, 1982.

[19] Neil Immerman. Languages that capture complexity classes. *SIAM J. Comput.*, 16(4):760–778, 1987.

[20] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, 2008.

[21] Elena Pezzoli. Computational complexity of Ehrenfeucht-Fraïssé games on finite structures. In *Proc. of CSL'98*, pages 159–170, 1998.

[22] Oleg Pikhurko and Oleg Verbitsky. Logical complexity of graphs: a survey. *CoRR*, abs/1003.4865, 2010.

[23] Moshe Y. Vardi. The complexity of relational query languages. In *Proc. of STOC'82*, pages 137–146, 1982.