Higher-Order Matching, Games and Automata

Citation for published version:

Digital Object Identifier (DOI):
10.1109/LICS.2007.23

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Higher-Order Matching, Games and Automata

Colin Stirling
School of Informatics
University of Edinburgh
cps@inf.ed.ac.uk

Abstract

Higher-order matching is the problem given $t = u$ where $t, u$ are terms of simply typed $\lambda$-calculus and $u$ is closed, is there a substitution $\theta$ such that $t\theta$ and $u$ have the same normal form with respect to $\beta\eta$-equality: can $t$ be pattern matched to $u$? This paper considers the question: can we characterize the set of all solution terms to a matching problem? We provide an automata-theoretic account that is relative to resource: given a matching problem and a finite set of variables and constants, the (possibly infinite) set of terms that are built from those components and that solve the problem is regular. The characterization uses standard bottom-up tree automata.

1. Introduction

Higher-order matching is the problem given $t = u$ where $t, u$ are terms of simply typed $\lambda$-calculus and $u$ is closed, is there a substitution $\theta$ such that $t\theta$ and $u$ have the same normal form with respect to $\beta\eta$-equality: can $t$ be pattern matched to $u$? The problem was conjectured to be decidable by Huet [5]. Loader showed that it is undecidable when $\beta$-equality is the same normal form by encoding the undecidable problem of $\lambda$-definability as matching [8]: also see [6] for a proof that using the halting problem.

In previous work, we confirm Huet’s conjecture [17]: a full (and very complicated) proof is in the long version of [17] which is not yet published. It first appeals to Padovani’s and Schubert’s reduction of matching to the conceptually simpler (dual) interpolation problem [12, 10]. It is then inspired by model-checking games (such as in [15]) where a model, a transition graph, is traversed relative to a property and players make choices at appropriate positions. We define a game where the model is a closed $\lambda$-term $t$ and play moves around it relative to a (dual) interpolation problem $P$. The game captures the dynamics of $\beta$-reduction on $t$ without changing it (using substitution). Unlike standard model-checking games, play may arbitrarily jump around a term because of binding. The principal virtue of the game is that small pieces of a solution term can be understood in terms of their subplays and how they, thereby, contribute to solving the problem $P$. Simple transformations on terms are defined and combinatorial properties shown. Decidability of matching follows from the small model property: if there is a solution to a problem then there is a small solution to it. The proof of this property uses “unfolding” a $\lambda$-term with respect to game playing, analogous to unravelling a transition system in modal logic, followed by refolding.

In this paper our interest is with a different, although related, question: can we independently characterize the set of all solution terms to an interpolation problem? Part of the hope is that this may lead to a simpler proof of decidability of matching. Again, we start with the term checking game. However, we slightly reformulate it and show that it underpins an automata-theoretic characterization relative to resource: given a problem $P$, a finite set of variables and constants the (possibly infinite) set of terms that are built from those components and that solve $P$ is regular. The characterization uses standard bottom-up tree automata. The states of the automaton are built from abstractions of sequences of moves in the game. The automaton construction works for all orders. Comon and Jurski define tree automata that characterize all solutions to a $4$th-order problem [2]. The states of their automata appeal to Padovani’s observational equivalence classes of terms [10]. To define the states of their automata at higher-orders, one would need to solve the problem of how to quotient the potentially infinite set of terms into their respective finite observational equivalence classes: however, as Padovani shows this problem is, in fact, equivalent to the matching problem itself. Ong shows decidability of monadic second-order logic of the tree generated by an arbitrary higher-order scheme [9]. The proof uses a game-semantic characterization of a scheme as an infinite $\lambda$-term. A property, expressed as an alternating parity tree automaton, of the tree has to be transferred to the infinite term. A key ingredient of the transition from game to automaton is Ong’s abstraction “variable profile” that captures a sequence of back-and-forth play jumping in a term which
is also central here.

In Section 2 we describe matching and how it reduces to the interpolation problem. In Section 3 we introduce tree automata and how they may be used to define well-formed \(\lambda\)-terms. The interpolation game is defined in Section 4 and the automata-theoretic account with the main result is presented in Section 5. Finally, there are concluding comments in Section 6.

### 2 Matching and interpolation

Assume simple types that are generated from a single base type \(0\) using the binary \(\to\) operator. A type is \(0\) or \(A \to B\) where \(A\) and \(B\) are types. If \(A \neq 0\) then it has the form \(A_1 \to \ldots \to A_n \to 0\), assuming \(\to\) associates to the right, written \((A_1, \ldots, A_n, 0)\) following Ong [9]. A standard definition of order is: the order of \(0\) is 1 and the order of \((A_1, \ldots, A_n, 0)\) is \(k + 1\) where \(k\) is the maximum of the orders of the \(A_i\).

Terms of the simply typed \(\lambda\)-calculus are built from a countable set of typed variables \(x, y, \ldots\) and constants \(a, f, \ldots\) (each variable and constant has a unique type). The smallest set \(T\) of simply typed terms is: if \(x(f)\) has type \(A\) then \(x : A \in T\) if \(t : B \in T\) and \(x : A \in T\) then \(\lambda x.t : A \to B \in T\); if \(t : A \to B \in T\) and \(u : A \in T\) then \(tu : B \in T\). The order of a typed term is the order of its type. A typed term is closed if it does not contain free variables. Throughout, we assume the definitions of \(\alpha\)-equivalence, \(\beta\) and \(\eta\)-reduction.

**Definition 1.** A matching problem is an equation \(v = u\) where \(v, u : 0\) and \(u\) is closed. The order of the problem is the maximum of the orders of the free variables \(x_1, \ldots, x_n\) in \(v\). A solution is a sequence of closed terms \(t_1 \ldots t_n\) such that each \(t_i\) has the same type as \(x_i\) and \(v(t_1/x_1) \ldots (t_n/x_n) = \beta\) \(u\).

Given a matching problem \(v = u\), the decision question is whether it has a solution: can \(v\) be pattern matched to \(u\)?

In the following we assume that all terms in normal form are in \(\eta\)-long form: if \(t : 0\) then \(t = u : 0\) where \(u\) is a constant or a variable, or \(u t_1 \ldots t_k\) where \(u : (B_1, \ldots, B_k, 0)\) is a constant or a variable and each \(B_i\) is in \(\eta\)-long form; if \(t : (A_1, \ldots, A_n, 0)\) then \(t = \lambda y_1 \ldots y_n t'\) where each \(y_i : A_i\) and \(t' : 0\) is in \(\eta\)-long form. Throughout, we write \(\lambda z_1 \ldots z_n\) for \(\lambda z_1 \ldots \lambda z_n\). A term is well-named if each occurrence of a variable \(y\) within a \(\lambda\)-abstraction is unique. Because of normal forms, \(\beta\)-equality and \(\beta\) \(\eta\)-equality coincide.

**Definition 2.** Assume \(u : 0\) and each \(v_i : A_i, 1 \leq i \leq n\), is a closed term in normal form and \(x : (A_1, \ldots, A_n, 0)\). An interpolation problem \(P\) has the form \(x v_1 \ldots v_n = u\). The type of problem \(P\) is that of \(x\) and the order of \(P\) is the order of \(x\). A solution of \(P\) of type \(A\) is a closed term \(t : A\) in normal form such that \(v_1 \ldots v_n = \beta\) \(u\). We write \(t \models P\) if \(t\) is a solution of \(P\).

Conceptually, interpolation is simpler than matching because of its single variable. If \(v = u\) is a matching problem with free variables \(x_1 : A_1, \ldots, x_n : A_n\) where \(v\) and \(u\) are in normal form, then its associated interpolation problem is \(x(x_1 \ldots x_n, v) = u\) where \(x : ((A_1, \ldots, A_n, 0), 0)\). This appears to raise order by 2 as with the reduction of matching to pairs of interpolation equations in Schubert [12]. However, we only need to consider potential solution terms (in normal form with the right type) \(\lambda z : t_1 \ldots t_n\) where each \(t_i : A_i\) is closed and so cannot contain \(z\): we say that such terms are canonical. Padovani provides a reduction of matching to dual interpolation that exactly preserves order [10]. A dual interpolation problem consists of a finite family of interpolation equations \(x v_1' \ldots v_k' = u_i\) and a finite family of disequations \(x v_1'' \ldots v_k'' \neq u_j\) all with the same free variable \(x\): a solution is a term \(t\) in normal form such that for each equation \(tv_1' \ldots v_k' = \beta\) \(u_i\) and for each disequation \(tv_{1''} \ldots v_{k''} \neq \beta\) \(u_j\). Dual interpolation underpins a notion of observational equivalence on terms that is used by Padovani to solve 4th-order matching [10].

**Fact 1.** A matching problem has a solution iff its associated interpolation problem has a canonical solution.

Consequently, the higher-order matching problem is equivalent to the decision question: given an associated interpolation problem \(P\), is there a canonical term \(t \models P\)? In the following we, therefore, concentrate on solutions to interpolation problems.

**Example 1.** \(x_1(\lambda z : x_1(\lambda z' : z a)) = a\) from [2] is a 4th-order matching problem where \(z, z' : (0, 0)\) and \(x_1 : (((0, 0), 0), 0)\). Its associated interpolation problem is \(x(\lambda x_1 : (\lambda x_1 : (\lambda z' : z a)) = a\) with \(x : (((0, 0), 0), 0), 0)\). A canonical solution has the form \(\lambda x : (\lambda y_1 : (\lambda y_1' \ldots y(\lambda y_k = w) \ldots ))\) where \(w\) is the constant \(a\) or one of the variables \(y_1'\), \(1 \leq j \leq k\). A solution is a term \(t : 0\) where \(u = f\) \(\lambda x : (\lambda y_1 : (\lambda y_1' \ldots y(\lambda y_k = w) \ldots ))\) assuming \(x_2 : 0\).

**Example 3.** \(x(\lambda z : z) = w\) where \(w = f(\lambda x_1 : (x'_1 : x'_2 : x'_3 : x'_4(x'_5))a\) also has order 3 where \(x\) has type \((0, 0), 0\) and \(f : ((0, 0), 0, 0, 0)\) assuming \(x_2 : 0\).

**Example 4.** \(x(\lambda y_1 : y_2 : y_1 : (\lambda y_3 : y_2 : (y_1 : (\lambda y_4 : y_1)))) = u\) where \(u = h(\lambda h(\lambda h))\), due to Luke Ong, is 5th-order with \(x : ((0, 0), 0), 0, 0, 0, \beta, 0)\).

Example 3 illustrates that a right term \(u\) of an interpolation problem may contain bound variables. Let \(X^t =\)
Definition 3. The ground closure of a closed term \( w \), whose bound variables belong to \( X' \) with respect to \( C \), written \( Cl(w, X', C) \), is defined inductively as follows: if \( w = a : 0 \) then \( Cl(w, X', C) = \{ a \} \); if \( w = f w_1 . . . w_n \) then \( Cl(w, X', C) = \{ w \} \cup Cl(w_i, X', C) \); if \( w = \lambda x'_1 . . . x'_n . a \) then \( Cl(w, X', C) = Cl(u(x'_1/x'_j, . . . , x'_n/x'_j), X', C) \) where \( x'_j \in X' \) and \( c_j \in C \) is the corresponding constant.

The ground closure of \( w \) of Example 3 with respect to \( \{ c_1, c_2, c_3 \} \) is \( \{ w, c_1(c_3), c_3, a \} \). The ground closure of \( u \) of Example 4 with respect to the empty set is its subterms \( \{ u, g(h(h(a))), h(h(a)), (a, a) \} \).

Definition 4. Given the problem \( P \) with equation \( x_1 . . . v_n = u \), let \( X_P \) be the (possibly empty) set of bound variables in \( u \) and \( C_P \) be a corresponding set of new constants (that do not occur in \( P \)), the forbidden constants. The right subterms are \( R_P = Cl(u, X_P, C_P) \).

We are interested in the set of closed terms \( t \) in normal form such that \( t \models P \) and \( t \) does not contain forbidden constants (belonging to \( C_P \)).

In the literature there are slight variant definitions of matching. Statman describes the problem as a range problem [14]: given \( v : (A_1, . . . , A_n, B) \) and \( u : B \) where both \( u \) and \( v \) are closed, are there terms \( t_1 : A_1, . . . , t_n : A_n \) such that \( v t_1 . . . t_n = \beta n u \)? If \( B = (A_1, . . . , A_m, 0) \) is of higher type then \( u \) in normal form is \( \lambda x'_1 . . . x'_m . w \). Therefore, we can consider the matching problem \( \langle (w x_1 . . . x_n) c_1 . . . c_m = w [c_1/x'_1, . . . , c_m/x'_m] \rangle \) where the \( c_i \)'s are forbidden constants (and cannot occur in a solution term). In [10] a matching problem is a family of equations \( v_1 = u_1, . . . , v_m = u_m \) to be solved uniformly: therefore, they reduce to a single equation \( f v_1 . . . v_n = f u_1 . . . u_m \) where \( f \) is a constant of the appropriate type.

3 Interpolation trees and tree automata

Given a potential solution term \( t \) in normal form to the interpolation problem \( P, x_1 . . . v_n = u \), there is the tree in Figure 1. If \( x : (A_1, . . . , A_n, 0) \) then the explicit application operator \( @ : ((A_1, . . . , A_n, 0), A_1, . . . , A_n, 0) \) has its expected meaning: \( @ t v_1 . . . v_n = t v_1 . . . v_n \). The terms \( t \) and \( v_j, 1 \leq j \leq n \), do not contain forbidden constants. These terms \( t \) are represented as labelled trees, tree \( t' \). If \( t' \) is \( g \ : \ 0 \) or \( a : 0 \) then tree \( t'' \) is the single node labelled with \( t' \). In the case of \( w u_1 . . . w_k \) when \( u \) is a variable, a constant or the initial \( 0 \), we assume that a dummy \( \lambda \) with the empty sequence of variables is placed before any subterm \( w_1 : 0 \) in its tree representation. With this understanding, if \( t' \) is \( w u_1 . . . w_k \) then tree \( t'' \) consists of the root node labelled \( u \) and \( k \)-successor nodes labelled with tree \( (w_i) \). We use the notation \( u \downarrow \) \( t' \) to represent that tree \( t' \) is the \( i \)-th successor of the node \( u \). If \( t' \) is \( \lambda \gamma . v \), where \( \gamma \) is possibly the empty sequence of variables, then tree \( t'' \) consists of the root node labelled \( \lambda \gamma \) and a single successor node tree node \( t(v) \). In the following we use \( t' \) to be the \( \lambda \)-term \( t' \), its \( \lambda \)-tree or the label (a constant, variable, initial \( \@ \) or \( \lambda \gamma \)) at its root node. For instance, the tree of the equation of Example 2 with solution term \( \lambda x_1 x_2 . x_1 (x_2 (x_1 a b b)) b \) and without explicit indices on edges is depicted in Figure 2: for instance, (2) \( \downarrow 1 \) (3) and (2) \( \downarrow 2 \) (15).

Given problem \( P, x_1 . . . v_n = u \), ideally we would like there to be a tree automaton that accepts exactly all instances of \( t \) in Figure 1 that solve \( P \): that is, we want an automaton that accepts \( \@ t v_1 . . . v_n \) if, and only if, \( t \models P \).

Definition 5. Assume \( \Sigma \) is a finite graded alphabet where each element \( s \in \Sigma \) has an arity \( ar(s) \geq 0 \). A \( \Sigma \)-tree is a finite tree where each node is labelled with an element of \( \Sigma \). If node \( n \) in \( t \) is labelled with \( s \) and \( ar(s) = k \) then \( n \) has precisely \( k \) successors in \( t \). Let \( T_\Sigma \) be the set of \( \Sigma \)-trees.

Definition 6. A \( \Sigma \)-tree automaton \( A = (Q, \Sigma, F, \Delta) \) where \( Q \) is a finite set of states, \( \Sigma \) is the finite alphabet, \( F \subseteq Q \) is the set of final states and \( \Delta \) is a finite set of transition rules \( s q_1 . . . q_k \Rightarrow q \) where \( k \geq 0, s \in \Sigma, ar(s) = k \) and \( q_1, . . . , q_k \) label the \( k \) successors of \( s \) (in correct order) then \( n \) is labelled \( q \). \( A \) accepts the \( \Sigma \)-tree \( t' \) if there is a run of \( A \) on \( t \) such that the root node of \( t \) is labelled with a final state, \( q \in F \). Let \( T_\Sigma (A) \) be the set of \( \Sigma \)-trees accepted by \( A \).

A \( \Sigma \)-tree automaton \( A \) involves a finite set of states \( Q \) and transitions \( \Delta \) (which can be nondeterministic). A run of \( A \) on the \( \Sigma \)-tree \( t \) is a labelling of it with elements of \( Q \) that is defined bottom-up, starting from the leaves of \( t \). If a node \( n \) of \( t \) is labelled \( s \in \Sigma \), \( ar(s) = k \geq 0 \), \( s q_1 . . . q_k \Rightarrow q \in \Delta \) and \( q_1, . . . , q_k \) label the \( k \) successors of \( s \) (in correct order) then \( n \) is labelled \( q \). \( A \) accepts the \( \Sigma \)-tree \( t \) if there is a run of \( A \) on \( t \) such that the root node of \( t \) is labelled with a final state, \( q \in F \). Let \( T_\Sigma (A) \) be the set of \( \Sigma \)-trees accepted by \( A \).
Figure 2. Example 2 with solution term

$k$ successors of $n$ (in correct order) then $n$ can be labelled $q$. This is repeated until no further nodes can be labelled. A run is then accepting if the root of $t$ is labelled with a final state, $q \in F$.

We state some pertinent properties of tree automata [1].

**Proposition 1.** The emptiness problem, given a $\Sigma$-tree automaton $A$ is $T_\Sigma(A) = \emptyset$, is decidable in linear time. For each $\Sigma$-tree automaton $A$ there is a $\bar{A}$ such that $T_\Sigma(\bar{A}) = T_\Sigma(A)$. For $\Sigma$-tree automata $A_1, A_2$ there is an $\bar{A}$ with $T_\Sigma(\bar{A}) = T_\Sigma(A_1) \cap T_\Sigma(A_2)$.

Consider a potential solution term $t$ to Example 2 such as the term rooted at (1) in Figure 2. Up to $\alpha$-equivalence, $t$ is $\lambda x_1 x_2 : w$ with $x_1 : (0, 0, 0)$ and $x_2 : (0, 0)$. What are its possible components (represented as a tree with dummy lambdas)? First there can be occurrences of $x_1$ with arity 2 and $x_2$ with arity 1. There can also be constant occurrences $a : 0$, of arity 0, $f$ of arity 2 and other constants. It suffices just to allow one other constant $b : 0$ (because constants other than $f$ and $a$ cannot contribute to a solution of this interpolation problem). Therefore, any possible $\lambda x_1 x_2 : w$ (with this restriction on constant occurrences) as a tree belongs to $T_\Sigma$ when $\Sigma = \{\lambda x_1 x_2, x_1, x_2, \lambda, f, a, b\}$.

There are $\Sigma$-trees, such as $x_1 (\lambda x_1 x_2 : w) a$, that are not the trees of well-formed terms in $T$. However, it is straightforward to define a tree automaton that accepts exactly the well-formed typed $\lambda$-terms (trees) over $\Sigma$ of the form $\lambda x_1 x_2 : w$. The idea is to internalize the finite alphabet $\Sigma$ as states. The automaton $A$ is $Q = \{[s] \mid s \in \Sigma\}, F = \{[\lambda x_1 x_2]\}$ and $\Delta$ consists of $a \Rightarrow [a], b \Rightarrow [b], \lambda \Rightarrow [\lambda]$ for any $s \in \Sigma - \{\lambda, \lambda x_1 x_2\}, \lambda x_1 x_2 [s] \Rightarrow [\lambda x_1 x_2]$ for any $s \in Q - \{\lambda, \lambda x_1 x_2\}, x_1 [\lambda] \Rightarrow [x_1], x_2 [\lambda] \Rightarrow [x_2], f [\lambda] \Rightarrow [f]$. The automaton could be extended to recognise all well-formed instances of the interpolation equation, such as Figure 2, by adding the elements $\emptyset, \lambda y_1 y_2, y_1, \lambda y_3, y_3$ to $\Sigma$.

The 5th-order equation in Example 4 has a solution term in Figure 3. A potential solution has the form $\lambda z : w$ with $z : ((0, 0), (0, 0), 0)$. Consider its possible components. For constants, we have $a, b, g$ and $h$ (the latter two of arity 1). A $z$ component has two successors $\lambda x_1, \lambda x_2$ where $x_1 : (0, 0)$ and $x_2 : 0$ which may bind variable occurrences $x_1$ with arity 1 and $x_2$ with arity 0. The problem is that the set of well-formed terms requires an infinite alphabet $\Sigma$ (even up to $\alpha$-equivalence) because of the family $\lambda z : ((\lambda x_1 z (\lambda x_2 z (\lambda x_3 z (\ldots z (\lambda x_k z (\ldots))))u_k)\ldots)u_1$ of terms as $k$ increases: this is also described in [2].

If we cannot define a tree automaton that accepts the well-formed terms of type $A$ of order 5 and above (modulo the restriction on constants) then it is extremely unlikely

---

1This can be exactly formalised using the game-theoretic characterisation of interpolation of Section 4.
that we could define an automaton that accepts just the solutions to an interpolation problem of type $\lambda$. Although there is active research extending automata on words and trees to infinite alphabets which preserve “good” properties, such as decidability of non-emptiness, see [11] for a recent survey, the results do not apply to the case caused by higher-order binding.

Therefore, we restrict the overall goal of seeking an automaton that can accept the solutions to a problem $P$. First, it is clear that given a type $\lambda$ and the finite alphabet $\Sigma$, there is a tree automaton that accepts the well-formed $\Sigma$-terms. Similarly for matching, we take as input not only $P$ but also the finite alphabet $\Sigma$. The main result, Theorem 2, is that there is a tree automaton that accepts exactly the set of well-formed $\Sigma$-terms that solve $P$. Therefore, relative to a finite alphabet $\Sigma$, the set of solutions to a problem $P$ is regular.

**Example 5.** In the case of Example 4 we can define a tree automaton that recognizes the set of well-formed terms (trees) of the form $\lambda z.w$ when $\Sigma$ is the set $\{\lambda z, \lambda z_1, \lambda z_2, \lambda x_1, \lambda x_2, \lambda, z, z_1, z_2, x_1, x_2, a, b, g, h\}$ with arities as in Figure 3 and $\Sigma_2 = \{z, z_1, z_2, x_1, x_2\}$. The idea is to internalize the syntax and also to include a finite memory as to which binders are still outstanding. $Q = \{[s,S] | s \in \Sigma, S \subseteq \Sigma_1\}$, $F = \{[\lambda z, \emptyset]\}$ and $\Delta$ contains rules such as: $x_2 \Rightarrow [x_2, \{x_2\}]$ (representing that having seen $x_2$ there is an outstanding $\lambda x_2$ binder), $z[\lambda z_1, S_1] [\lambda z_2, S_2] \Rightarrow [z, S_1 \cup S_2 \cup \{z\}]$, $\lambda x_2[\lambda z, S] \Rightarrow [\lambda x_2, S - \{x_2\}]$ for $s \in \{z, z_1, z_2, x_1, x_2, a, b, g, h\}$. Again, we can extend the automaton to accept all well-formed instances of the interpolation problem in Figure 3 by extending $\Sigma$.

Comon and Jurski define tree automata that characterize all solutions to a 4th-order problem [2]. In Section 8 of their paper they make it clear that there are two problems with extending their automata beyond the 4th-order case. The first is that the states of the automaton are constructed out of observational equivalence classes of terms due to Padovani [10]. Up to a 4th-order problem, one only needs to consider finitely many terms. With 5th and higher orders, this is no longer true and one needs to quotient the potentially infinite terms into their respective observational equivalence classes in order to define only finitely many states: however as Padovani shows this procedure is, in fact, equivalent to the matching problem itself [10]. The second problem they mention and illustrate with an example is that fifth-order terms may (essentially) contain infinitely many different variables as discussed here. We are able to overcome the first problem but not the second. Our solution is indirect and proceeds via a game-theoretic characterisation of matching, Theorem 1. It underlies a more elementary understanding of $\beta$-reduction in terms of families of sequences of moves which can then be abstracted into structures that
will become states of an automaton.

4 Interpolation games

We are going to define a game on interpolation trees such as Figure 1 which is now the arena. Games have been a key ingredient in our previous work on matching [16, 17]. The aim is to account for the dynamics, $\beta$-reduction, without changing terms by using substitution. It turns out that it could also be described using game-semantics following Ong [9]. Indeed, we slightly reformulate previous versions of the game so that it is closer to game-semantics. However, as in [17], we avoid questions, answers and their justification pointers by appealing to look-up tables.

Assume $P$ is the problem $x_1 \ldots x_n = u$, $t$ is a potential solution term and assume the sets $R = R_P$ and $C = C_P$. We define the game $G(t, P)$ played by one participant, player $\forall$, the refuter who attempts to show that $t$ is not a solution of $P$.

Definition 8. $N$ is the set of nodes of the interpolation tree $\Theta_j$ where $j \geq 0$, is the set \{$x \mid x \in R \cup \{[\forall, \exists]\}$\} of game-states. $[\forall]$ and $[\exists]$ are the final game-states. $V$ is the set of game-states that occur in $t$ and in $v_1, \ldots, v_n$. For each $i \geq 1$, the set of look-up tables $\Theta_i$ is iteratively defined: $\Theta_1 = \{\theta_1\}$ where $\theta_1 = \emptyset$ and $\Theta_{i+1}$ is the set of partial maps $V \rightarrow (N \times \bigcup_{j \geq i} \Theta_j) \cup C$.

Definition 9. A play of $G(t, P)$ is a finite sequence of positions $t \forall q_1 \theta_1, \ldots, t \forall q_n \theta_n$, where each $t_i \in N$, each $q_i \in S$ and $q_n$ is final and each $\theta_i \in \Theta_i$ is a look-up table. For the initial position $t_1 = \emptyset$ where $\emptyset$ is the root of the interpolation tree, $q_1 = [u]$ where $u$ is the right term of $P$ and $\theta_1$ is the empty look-up table. Player $\forall$ loses the play if the final state is $[\exists]$, otherwise she wins the play.

The central feature of a play of $G(t, P)$ is that control jumps from nodes of $t$ to nodes of the $v_j$’s and back again repeatedly \(^2\). The game appeals to a finite set of states $S$ comprising goal states $[r], r \in R$, and final states, $[\forall]$, winning for the refuter, and $[\exists]$, losing for the refuter. As play proceeds by moving about $t$ and the $v_j$’s of Figure 1 one needs an account of the meaning of the current free variables. A free variable in a subtree of $t$ is either associated with a subtree of a $v_j$ or with a forbidden constant. Similarly, a free variable in a subtree of $v_j$ is associated with a subtree of $t$ or with a forbidden constant. So, the game appeals to the look-up tables $\theta_k \in \Theta_k, k \geq 1$: $\theta_k$ is a partial map from variables to forbidden constants $C$ or to pairs $A$. $t_m = \emptyset : A, A = (A_1, \ldots, A_n, 0), A_j, A_n, 0)$, for $i : 1 \leq i \leq n + 1$, $t_m \downarrow t_i'$ and $t_1' = \lambda x_1 \ldots x_n$. Then $t_{m+1} = t_1', q_{m+1} = q_m$ and $\theta_{m+1} = \theta_m(t_1' \theta_m / x_1, \ldots, t_n' \theta_m / x_n)$.

B. $t_m = \lambda \forall t'$. Then $t_{m+1} = t', q_{m+1} = q_m$ and $\theta_{m+1} = \theta_m$.

C. $t_m = : 0$. Then $t_{m+1} = t_m$ and $\theta_{m+1} = \theta_m$. If $q_m = [a]$ then $q_{m+1} = [\exists]$ else $q_{m+1} = [\forall]$.

D. $t_m = f : (A_1, \ldots, A_n, 0), n > 0$ and for each $i : 1 \leq i \leq n, t_m \downarrow t_i'. If q_m \neq \{f w_1 \ldots w_n\}$ then $t_{m+1} = t_m, q_{m+1} = [\forall]$ and $\theta_{m+1} = \theta_m$. Otherwise $\forall$ chooses a direction $j : 1 \leq j \leq n$ and $t_{m+1} = t_j'$. If $A_j = 0$ then $q_{m+1} = [w_j]$ and $\theta_{m+1} = \theta_m$. Otherwise, $w_j = \lambda x_1^i \ldots x_k^i \cdot w$ and $t_{m+1} = \lambda z_1 \ldots z_k$. So, $q_{m+1} = [w(c_1 / x_1^i, \ldots, c_k / x_k^i)]$ and $\theta_{m+1} = \theta_m(c_1 / z_1, \ldots, c_k / z_k)$.

E. $t_m = y : o$ and $\theta_m(y) = t' \theta_k$. Then $t_{m+1} = t'$, $q_{m+1} = q_m$ and $\theta_{m+1} = \theta_k$.

F. $t_m = y : 0$ and $\theta_m(y) = c$. Then $t_{m+1} = t_m$ and $\theta_{m+1} = \theta_m$. If $q_m = [c]$ then $q_{m+1} = [\exists]$ else $q_{m+1} = [\forall]$.

G. $t_m = y : (A_1, \ldots, A_n, 0), n > 0, \theta_m(y) = t' \theta_k, t' = \lambda z_1 \ldots z_n$ and for each $i : 1 \leq i \leq n, t_m \downarrow t_i'$. Then $t_{m+1} = t', q_{m+1} = q_m$ and $\theta_{m+1} = \theta_k(t_i' \theta_m / z_1, \ldots, t_n' \theta_m / z_n)$.

H. $t_m = y : (A_1, \ldots, A_n, 0), n > 0, \theta_m(y) = c$ and for each $i : 1 \leq i \leq n, t_m \downarrow t_i'$. If $q_m \neq \{c w_1 \ldots w_n\}$ then $t_{m+1} = t_m, q_{m+1} = [\forall]$ and $\theta_{m+1} = \theta_m$. Otherwise $\forall$ chooses a direction $j : 1 \leq j \leq n$ and $t_{m+1} = t_j'$. If $A_j = 0$ then $q_{m+1} = [w_j]$ and $\theta_{m+1} = \theta_m$. Otherwise, $w_j = \lambda x_1^i \ldots x_k^i \cdot w$ and $t_{m+1} = \lambda z_1 \ldots z_k$. So, $q_{m+1} = [w(c_1 / x_1^i, \ldots, c_k / x_k^i)]$ and $\theta_{m+1} = \theta_m(c_1 / z_1, \ldots, c_k / z_k)$.

Figure 4. Game moves


\[ t' \theta_j \] where \( t' \) is a subtree and \( \theta_j \) a previous look-up table, so \( j < k \). A variable \( y \) in \( t \) (or \( e \)) may be associated with a subtree \( t' \) of \( \theta_j \) (or \( t \)) which itself may contain free variables: hence, the need for \( \theta_1(y) \) to be a pair \( t' \theta_j \) as \( \theta_j \) records the values of the free variables in \( t' \) at the earlier position.

**Definition 10.** If the current position in \( G(t, P) \) is \( t_m q_m \theta_m \) and \( q_m \) is not final then the next position \( t_{m+1} q_{m+1} \theta_{m+1} \) is determined by a unique move in Figure 4.

At the initial \( \theta_0 \) play proceeds to the first successor subtree \( \lambda x_1 \ldots x_n \) and the look-up table is updated accordingly (each of the other successor subtrees \( t'_{i+1} \) with the empty look-up table \( \theta_1 \) is associated with \( x_i \): if later play reaches a variable occurrence \( x_i : A_i \), then it jumps to \( t'_{i+1} : A_i \)). We assume standard updating notation for \( \theta_{m+1} : \gamma(\delta_1/y_1, \ldots, \delta_m/y_m) \) is the partial function similar to \( \gamma \) except that \( \gamma(y_i) = \delta_i \). If play is at \( \lambda y \), where \( y \) can be empty, then it descends the tree. At a constant \( a : 0 \), the refuser loses if the goal state is \([a] \) and wins otherwise. At a constant \( f \), with arity more than 0, \( \forall \) immediately wins if the goal state is not of the form \([f w_1 \ldots w_n] \). Otherwise \( \forall \) chooses a successor \( j \) and play moves to the \( j \)th successor of (the node labelled) \( f \), the \( t' \) such that \( f \downarrow_j t' \). If \( w_j = 0 \) then the goal state is \([w_j] \). However, if \( w_j \) is higher-order then forbidden constants are introduced and the look-up table is updated accordingly. If play is at a variable \( y : 0 \) and its entry in the current look-up table is \( t' \theta_h \) then play jumps to \( t' \) and \( \theta_h \) becomes the look-up table. If the entry for \( y \) is a forbidden constant then a final state will be reached. If \( y \) is higher-order and its entry is \( t' \theta_k \) where \( t' = \lambda z_1 \ldots z_n \) then play jumps to \( t' \) and the look-up table is \( \theta_1 \) together with the association of \( t' \theta_m \) for \( z_i \) when \( y \downarrow_i t' \), where \( i : 1 \leq i \leq n \): this illustrates how back-and-forth play jumping captures \( \beta \)-reduction as the occurrence of \( y \) is associated with \( t' \), and, therefore, occurrences of free \( z_i \) beneath \( t' \) are associated with \( y \)'s \( i \)th successor. The final case is when \( y \) is higher-order and its entry in the look-up table is a forbidden constant: again, there is the possibility that further forbidden constants will be introduced. Player \( \forall \) can only exercise choice at a higher-order constant whose arity is more than one (rules D and H) thereby carving out a branch of the right term \( u \) (modulo introduction of forbidden constants).

**Definition 11.** Player \( \forall \) loses the game \( G(t, P) \) if she loses every play and otherwise she wins the game.

The game characterizes interpolation which essentially follows from [16, 17].

**Theorem 1.** Player \( \forall \) loses \( G(t, P) \) if, and only if, \( t \models P \).

**Example 6.** A solution to Example 2 is depicted in Figure 2.

The play initially proceeds as follows.

\( (1) [\text{f aa}] \theta_1 = \theta_1 \) \( (2) [\text{f aa}] \theta_2 = \theta_2 \) \( (3) [\text{f aa}] \theta_3 = \theta_3 \) \( (4) [\text{f aa}] \theta_4 = \theta_4 \)

Play starts at node \( \theta_0 \) with goal state \([\text{f aa}] \) and descends to node (1) with the subtree at (17) associated with \( x_1 \) and the subtree at (19) with \( x_2 \). After play descends to (2), it jumps to (17) and then the subtree at node (3) is associated with \( y_1 \) and the subtree at (15) with \( y_2 \). So play descends to (18) and jumps to (3). The move to (4) results in a jump to (19) and then to (20). At which point there is a \( \forall \) choice, as to (21) or (23), both with the same reduced goal state \([a] \).

If the first choice is then play continues as follows.

\( (21) [a] \theta_{10} = \theta_{10} \) \( (22) [a] \theta_{11} = \theta_{10} \) \( (23) [a] \theta_{12} = \theta_{12} \) \( (24) [a] \theta_{13} = \theta_{13} \)

The other choice is similar as both choices lead to play returning to node (5) with the goal \([a] \). Play then descends to (6) and, therefore, jumps again to (17) but now with (7) associated with \( y_1 \) and (13) with \( y_2 \). From (18) it jumps to (7). Similarly, at (8) it jumps to (17) and returns to (9) without changing state \([a] \). Consequently, play ends at (10) and \( \forall \) loses.

**Example 7.** Assume \( P \) is Example 4 with a solution term \( t \) all depicted in Figure 3. The single play for \( G(t, P) \), presented in Figure 5, is quite intricate with significant jumping in the interpolation tree. For instance, at move 3 when at node (2) play jumps to (21), descends to (22) and jumps to (3) and then later at move 11 when at node (6) jumps to (23). Even later at move 61 at node (10) which like (6) is also bound by (3), play again returns to (23). Player \( \forall \) eventually loses when play reaches (12).

If \( t \models P \), so \( \forall \) loses the game \( G(t, P) \), then the total number of different plays is at most the number of branches in the right term \( u \) of \( P \). There are many interesting combinatorial properties of plays, some of which are briefly described in [17, 16]. For the proof of the main result, Theorem 2, the main feature is understanding the back-and-forth play jumping.

### 5 Interpolation tree automata

In this section we define tree automata for an interpolation problem \( P \) relative to the finite alphabet \( \Sigma \). Assume a fixed problem \( P \) with \( R = \Sigma \) and \( C = C_P \).
Definition 12. Assume an interpolation tree. Variable occurrence \( z \) (at node \( n \)) is an immediate descendent of \( u \) (at node \( m \)) if for some \( i \), \( u_i \sqsubseteq \lambda x_i \Phi_i \) and \( \lambda x_i \Phi_i \) binds \( z \) (at node \( n \)). Variable occurrence \( z \) is a descendent of \( u \) if \( z \) is an immediate descendent of \( u \) or variable occurrence \( y \) is an immediate descendent of \( u \) and \( z \) is a descendent of \( y \). Variable occurrence \( z \) is a descendent of a constant if there is a constant occurrence \( f \) and \( z \) is a descendent of \( f \).

The term \( \lambda x_1 y_1 y_2 (x_1 x_2) \) is a solution to Example 3. The single variable occurrences of \( x \) and \( z_2 \) are descendents of the constant \( f \). In the interpolation game, entries for such variable occurrences in a look-up table are forbidden constants. Therefore, without loss of generality, we assume that the set of variables \( V \subseteq \Sigma \) is partitioned into \( V_1 \) and \( V_2 \). \( V_1 \) contains variables whose occurrences must be descendents of constants and \( V_2 \) contains those that cannot be descendents of constants. We ensure that \( \Sigma \) contains enough variables to guarantee this.

In the game, play jumps around the interpolation tree as illustrated in Examples 6 and 7. The question is how to capture this jumping with a tree automaton. The solution is based on Ong [9] (which is a different setting, with a fixed infinite \( \lambda \)-term and an alternating parity automaton). We need to break apart the motion of jumping into and out of components into constituents. This we do using variable assumptions: Ong calls them “variable profiles” in his setting.

Definition 14. Assume \( P, x_1 v_1 \ldots v_n = u, \Sigma \) and the variables \( V \) partitioned into \( V_1 \cup V_2 \). The \( \Sigma \)-tree automaton for \( P \) is \( A_P = (Q, \Sigma; F, \Delta) \) where \( Q \) is the set of states of sets \( \{ \Gamma_1, \ldots, \Gamma_k \} \), \( k \geq 0 \), and \( F = \{ (u,0) \} \) and the transition relation \( \Delta \) is defined on nodes of the \( \Sigma \)-tree by cases on \( \Sigma \).

- \( a : 0. \) Then \( a \Rightarrow A \) where \( A \subseteq \{ (a,0) \} \).
- \( z : 0 \) and \( z \in V_1 \). Then \( z \Rightarrow \Lambda \subseteq \{ (a,0) \} \).

Figure 6. A \( \Rightarrow \) assumption for Example 3
\[
\{ (c_1, \{ (z, c_1) \}), \ldots, (c_k, \{ (z, c_k) \}) \} \text{ for } k \geq 0
\]
and \( c_1 : 0, \ldots, c_k : 0 \in C. \)

- \( z : 0 \) and \( z \in V_2. \) Then \( z \Rightarrow \Lambda \subseteq \{ (r_1, \{ (z, r_1) \}), \ldots, (r_m, \{ (z, r_m, 0) \}) \} \) for \( m \geq 0 \) and \( r_1, \ldots, r_m \in R. \)

- \( f : (A_1, \ldots, A_k, \emptyset) \) and \( f \Lambda_1 \ldots \Lambda_k \Rightarrow \Lambda \) when (1) and (2). (1). If \( (r, \Gamma) \in \Lambda \) then \( r = f w_1 \ldots w_k \) and for each \( i : 1 \leq i \leq k \) there is \( (r_i, \Gamma_i) \in \Lambda_i \) with \( w_i = \lambda x_i' \ldots x_{i_m}' w_i m_i \geq 0, f \downarrow, \lambda y_i' \ldots y_{i_m}', r_i = w_i' \{ c_1/x_i', \ldots, c_{i_m}/x_{i_m}' \} \) and \( \Gamma = \bigcup_{i \leq k} (\Gamma_i - \bigcup_{i \leq m_i} \{ \Gamma(y'_i) \}). \) (2). If \( (r_i, \Gamma_i) \in \Lambda_i \) for \( i : 1 \leq i \leq k \) then there is \( f w_1 \ldots w_k, \Gamma \in \Lambda \) with \( w_i = \lambda x_i' \ldots x_{i_m}' w_i m_i \geq 0, f \downarrow, \lambda y_i' \ldots y_{i_m}', r_i = w_i' \{ c_1/x_i', \ldots, c_{i_m}/x_{i_m}' \} \) and \( \Gamma_i - \bigcup_{i \leq m_i} \{ \Gamma(y'_i) \} \subseteq \Gamma. \)

- \( z : (A_1, \ldots, A_k, \emptyset) \) and \( z \in V_1. \) Then \( z \Lambda_1 \ldots \Lambda_k \Rightarrow \Lambda \) when (1) and (2). (1). If \( (r, \Gamma) \in \Lambda \) then \( r = c w_1 \ldots w_k, (z, r, c) \in \Gamma \) and for each \( i : 1 \leq i \leq k \) there is \( (r_i, \Gamma_i) \in \Lambda_i \) with \( w_i = \lambda x_i' \ldots x_{i_m}' w_i m_i \geq 0, z \downarrow, \lambda y_i' \ldots y_{i_m}', r_i = w_i' \{ c_1/x_i', \ldots, c_{i_m}/x_{i_m}' \} \) and \( \Gamma = \{ (z, r, c) \} \bigcup \{ c w_1 \ldots w_k, \Gamma \in \Lambda \} \subseteq \Gamma. \) (2). If \( (r_i, \Gamma_i) \in \Lambda_i \) for \( i : 1 \leq i \leq k \) then there is \( (r, \Gamma) \in \Lambda \) with \( r = c w_1 \ldots w_k, \lambda x_i' \ldots x_{i_m}' w_i m_i \geq 0, z \downarrow, \lambda y_i' \ldots y_{i_m}', r_i = w_i' \{ c_1/x_i', \ldots, c_{i_m}/x_{i_m}' \} \) and \( \Gamma_i - \bigcup_{i \leq m_i} \{ \Gamma(y'_i) \} \subseteq \Gamma. \)

- \( \lambda \). Then \( \lambda \Lambda \Rightarrow \Lambda \) for any \( \Lambda. \)

\( \lambda \) : \( (A_1, \ldots, A_n, A_0) \). Then \( \lambda \Lambda_1 \ldots \Lambda_n \Rightarrow \{ (u, 0) \} \) when (1). (1). \( \Lambda_1 = \{ (u, 0) \} \) and if \( (z, r', \Gamma) \in \Gamma \) then \( x_i \) for some \( i \) and \( r', \Gamma ' \) \in \( \Lambda_i \), and if \( (r_i, \Gamma_i) \in \Lambda_i \) then \( (x_i, r_i, \Gamma_i) \in \Gamma. \)

Example 8. Consider the automaton for Example 2. A potential solution term such as the one depicted in Figure 2 has the form \( \lambda x_1 x_2 w. \) So \( \Sigma = \{ a, b, f, x_1, x_2, y_1, y_3, \lambda, \lambda y_1, \lambda y_2, \lambda x_1 x_2 \}. \) The variables \( V \) are \( \{ x_1, x_2, y_1, y_3 \}, V_1 = 0 \) and \( V_2 = V. \) The set \( R = \{ f a a, a \}. \) A mode is \( (r, \Gamma) \) where \( r \in R \) and \( \Gamma \) is a set of variable assumptions. Next, we consider the transitions, first for constants.

\[
a \Rightarrow \Lambda \subseteq \{ (a, 0) \} \quad b \Rightarrow \emptyset \quad f \emptyset \Rightarrow \emptyset
\]

\[
f \{ (a, \Gamma_1) \} \Rightarrow \{ (f a a, \Gamma_1 \cup \Gamma_2) \}
\]

The last rule allows \( f \) to be labelled with the state \( \{ f a a, \Gamma_1 \cup \Gamma_2 \} \) when its descendents are labelled with the states \( \{ (a, \Gamma_1) \} \) and \( \{ (a, \Gamma_2) \}. \) Next we examine the rules for variables \( y_1 \) and \( y_3 \) which are both of type \( 0. \)

\[
y_1 \Rightarrow \Lambda \subseteq \{ (f a a, \{ (y_1, f a a, 0) \}), (a, \{ (y_1, a, 0) \}) \}
\]

\[
y_3 \Rightarrow \Lambda \subseteq \{ (a, \{ (y_3, a, 0) \}) \}
\]

It is unnecessary to include \( (f a a, \{ (y_3, f a a, 0) \}) \) as a mode because it could never appear within a state of an accepting run of the automaton. The rules for \( x_1 \) and \( x_2 \) are more interesting. Let \( A(x_1) = \{ x_1, f a a, \{ (y_1, f a a, 0) \}, A'(x_1) = \{ x_1, a, \{ (y_1, a, 0) \} \}) \) and \( A(x_2) = \{ x_2, f a a, \{ (y_3, a, 0) \} \}. \) Assume \( \Gamma' = \{ (A(x_1), A'(x_1), A(x_2)) \}. \)

\[
x_1 \emptyset \Rightarrow \emptyset \quad x_2 (a, \Gamma) \Rightarrow \{ (f a a, \Gamma \cup \{ A(x_2) \}) \} \subseteq \{ A(x_1) \}
\]

\[
x_1 (a, \Gamma) \Rightarrow \{ (a, \Gamma \cup \{ A'(x_1) \}) \} \subseteq \{ A(x_1) \}
\]

\[
x_1 (f a a, \Gamma) \Rightarrow \{ (f a a, \Gamma \cup \{ A(x_1) \}) \} \subseteq \Gamma'
\]

A variable occurrence \( x_1 \) can be labelled with the state \( \{ (f a a, \Gamma') \} \) provided that \( A(x_1) \in \Gamma' \) and its first successor is labelled \( \{ (f a a, \Gamma) \} \) such that \( \Gamma \cup \{ A(x_1) \} = \Gamma'. \) There are the rules for the \( \lambda \)'s and \( \emptyset. \)

\[
\Lambda \Lambda \Rightarrow \Lambda \quad \lambda x_1 x_2 \Lambda \Rightarrow \Lambda
\]

\[
\lambda y_1 y_2 \Lambda \Rightarrow \Lambda \quad \lambda y_3 \Lambda \Rightarrow \Lambda
\]

\[
\emptyset (f a a, \Gamma) \Rightarrow \Lambda \quad \emptyset (f a a, \emptyset) \text{ if (I)}
\]

where (I) is: \( (f a a) \in \Gamma \) if \( \{ (f a a, \{ (y_1, f a a, 0) \}) \} \) \in \( \Lambda_1, \) \( A'(x_1) \in \Gamma \) if \( \{ (a, (y_1, a, 0) \}) \) \in \( \Lambda_1 \) and \( A(x_2) \in \Gamma \) if \( \{ f a a, \{ (y_3, a, 0) \} \} \in \Lambda_2. \) Using these rules, it is easy to show that the tree of Figure 2 is accepted. The rules for \( x_1 \) allow “pumping” in the sense of [3]: the automaton accepts any term \( \lambda x_1 x_2 x_1 (x_1 \ldots x_1 (x_2 (x_1 \ldots x_1 a b) \ldots )b) \ldots )b. \)

The following is the main result of the paper.

Theorem 2. Assume \( P, x_1, \ldots, x_n = u, \Sigma \) and the \( \Sigma \)-tree automaton \( A_P. \) For any well-formed \( \Sigma \)-term \( t, A_P \) accepts the tree \( \emptyset t v_1 \ldots v_n \text{ if } t \vdash P. \)
6 Conclusion

It is unclear whether we can generalize Theorem 2 without developing automata for languages with infinite alphabets. In fact, there is the more specific problem of how to define automata in the presence of higher-order binding. For 4th-order problems, there is an automaton that recognizes its full set of solutions (up to α-equivalence) even though the syntax may be infinite. Comon and Jurski define a special kind of automata, □-automata, to achieve this [2]. The occurrence of a leaf □ in a term tree represents any (syntactically correct) subtree. A □ cannot contribute to the solution of the matching problem. Given a 4th-order problem \( P \), there is a finite alphabet \( \Sigma \) such that every solution \( t \) is recognized by a □-automaton (where □s of the correct type replace subtrees of \( t \) and all other nodes of \( t \) are labelled by elements of \( \Sigma \)). In our framework, we can also achieve this. Play in \( G(\xi, P) \) does not enter a subtree that is replaced by □. Consequently, given the set \( \Sigma \), we can assume the extra schematic automaton rule \( u \in \xi \Rightarrow \emptyset \) for all \( u \not\in \Sigma \).

What about decidability of higher-order matching? A corollary of Theorem 2 is that matching relative to resource is decidable. Indeed this is also true for \( \lambda \)-definability [7], even though without out this restriction the problem is undecidable. For higher-order matching, a proof of the small model property is, therefore, still needed. However given the property, there is now a more intelligent decision procedure using non-emptiness of tree automata. Moreover, it is possible that automata could underpin a more direct proof of this property. The thought is that given an interpolation equation \( xv_1 \ldots v_n = u, x : A = (A_1, \ldots, A_n, 0) \), there can only be finitely many “inequivalent” \( x_i : A_i \) variable assumptions because of their bounded depth irrespective of the number of different variables a term \( t : A \) may contain.

References


