Abstract. This paper presents Subsumption-oriented Push-Down Automata (SPDA), a very general stack formalism used to describe forest ("AND-OR" tree) traversals. These automata may be used for parsing or the interpretation of logic programs. SPDA allow a Dynamic Programming execution which breaks computations into combinable, sharable and storable sub-computations. They provide computation sharing and operational completeness and solve some of the problems posed by the usual depth-first, left-to-right traversals (as implemented in PROLOG). We give an axiomatization of SPDA and two examples of their use: the evaluation of logic programs and parsing with Tree Adjoining Grammars. SPDA may also serve in other areas such as Constraint Logic Programming, Abstract Interpretations, or Contextual parsing.

1 Introduction

The Push-Down Automaton (PDA) is a well-known machine that uses single stack operations for context-free parsing. B.Lang proposed in [Lan74] a technique extending Dynamic Programming (DP) to efficiently execute PDA, especially when they are non-deterministic. The idea consists in computing the complete set of stack tops that may appear in any calculation of the PDA. The reachable stacks, and especially success ones, can be extracted from this set of tops. Dynamic programming techniques are a good way to share sub-computations and are operationally complete.

Our purpose is to extend PDA to richer domains (Herbrand domain, constraint domains) while preserving the advantages of Dynamic Programming. These domains, in addition to a rich vocabulary, often provide a subsumption order which can be used to share computations. Dynamic Programming avoids redundancy by computing only the most general (sub-)goals. Furthermore, logic program non-determinism tends to duplicate sub-computations, making computation sharing even more interesting.

Besides computation sharing, Dynamic Programming also offers a solution to completeness problems one encounters with standard depth-first strategies (as in PROLOG). It improves termination by avoiding those loops which correspond to infinite recomputations. Our approach is related to tabulation [TS86, Vie87] or magic-set [Sek89] techniques proposed for Logic Programming, but encompasses them by being more flexible. Indeed, Dynamic Programming works for various automata which

* This work has been partly supported by the Eureka Software Factory (ESF) project.
can encode all kinds of resolution strategies (e.g. SLD, Bottom-Up, and Earley Deduction[Por86]). Extensions of tabulation or magic-set techniques have been proposed for constraint logic programming and abstract interpretation [KK90, Kan90].

The heart of this paper is devoted to a systematic axiomatization of a very general class of automata, namely Subsumption-oriented Push-Down Automata. These automata are conceived for a Dynamic Programming execution with subsumption. We give axioms to ensure the validity of such an execution.

Section 2 presents the Subsumption-oriented Push-Down Automata. In the last two sections, we give two concrete applications of these SPDA: pure Logic Programs interpretation and parsing with Tree Adjoining Grammars.

2 Subsumption-oriented Push-Down Automata

2.1 Preliminaries about Diagram Notations

In this paper, we use commutative diagrams to introduce the needed axioms and to ease proofs. These diagrams are widely used in papers which employ Category Theory. We recall basic reading conventions on the below diagram example.

\[
\begin{array}{c}
R_1 \\
\downarrow f \\
\downarrow c 
\end{array} \quad \text{R}_2 \quad \begin{array}{c}
v \\
\bar{v}
\end{array} \quad \begin{array}{c}
d
\end{array}
\]

An arrow \( a \xrightarrow{R_1} b \) states that the objects \( a \) and \( b \) are related by \( R_1 \) (\( a R_1 b \)).

An arrow \( a \xrightarrow{f} c \) where \( f \) denotes a (partial) function can be read "\( a \in \text{Dom}(f) \land c = f(a) \)". Non-oriented arrows (\( c \equiv d \)) are used for equivalence relations. Plain arrows denote universal quantified premises while dashed arrows denote existential quantified conclusions. Thus, the previous diagram should be read:

\[\forall a, b, c : \text{objects}, (a R_1 b \land a \in \text{Dom}(f) \land c = f(a)) \Rightarrow (\exists d : \text{object}, b R_2 d \land c \equiv d)\]

2.2 Ordered Stack Domains

Generally speaking\(^2\), a stack is only a finite sequence of elements of a domain \( D \). A stack \( \xi \) is noted (using a list notation) \([A_1, \ldots, A_n]\) where the top of the stack is on the left. The empty stack is noted []. We use the notation \([A_1, \ldots, A_n | \theta]\) to describe a stack where the \( n \) first top elements are \( A_1 \ldots A_n \) and the rest is the stack \( \theta \). \( \text{tail} \xi \) denotes the stack \( \xi \) minus its first top element and \( h(\xi) \) denotes the height of \( \xi \). We also introduce the cut operators \( \pi_n \) which return (when possible) the \( n \) first top elements.

\(^2\) We limit our presentation to the usual stack notion, but the interested reader can find in [BVdlC] an axiomatization which allows more exotic stacks and still preserves the results of this paper.