On Policy Reuse: An Expressive Language for Representing and Executing General Policies that Call Other Policies

Blai Bonet[1], **Dominik Drexler**[2], **Hector Geffner**[3,2]
[1]Universitat Pompeu Fabra, Spain
[2]Linköping University, Sweden
[3]RWTH Aachen University, Germany
bonetblai@gmail.com, dominik.drexler@liu.se, hector.geffner@ml.rwth-aachen.de

Abstract

Recently, a simple but powerful language for expressing and learning general policies and problem decompositions (sketches) has been introduced in terms of rules defined over a set of Boolean and numerical features. In this work, we consider three extensions of this language aimed at making policies and sketches more flexible and reusable: internal memory states, as in finite state controllers; indexical features, whose values are a function of the state and a number of internal registers that can be loaded with objects; and modules that wrap up policies and sketches and allow them to call each other by passing parameters. In addition, unlike general policies that select state transitions rather than ground actions, the new language allows for the selection of such actions. The expressive power of the resulting language for policies and sketches is illustrated through a number of examples.

Introduction

A new language for representing problem structure explicitly has been introduced recently in the form of *sketches* (Bonet and Geffner 2021, 2023). Sketches are collections of rules of the form $C \mapsto E$ which are defined over a set of Boolean and numerical domain features Φ, where C expresses Boolean conditions on the features, and E expresses qualitative changes in their values. Each sketch rule captures a subproblem: the problem of going from a state s whose feature values satisfy the condition C, to a state s' where the feature values change with respect to s in agreement with E. The language of sketches is powerful as it can encode everything from simple goal serializations to full general policies.

The width of a sketch for a class of problems bounds the complexity of solving the resulting subproblems (Bonet and Geffner 2021, 2023). For example, a sketch of width k decomposes problems into subproblems that are solved by the IW algorithm in time exponential in k (Lipovetzky and Geffner 2012). Sketches provide a direct generalization of policies, which are sketches of width 0 that result in subproblems that can be solved in a single step.

Combinatorial optimization methods for learning policies and sketches of this form has been developed by Francès, Bonet, and Geffner (2021) and Drexler, Seipp, and Geffner (2022) where both the rules and the features are obtained.

The current language of policies and sketches, however, does not support the *reuse* of other policies and sketches, and assumes instead that the top goals are set up externally.[1] Yet for *reusing* policies and sketches, goals must be set up internally as well; e.g., for reusing a general policy π_{on} that achieves the atom $on(x, y)$ for any blocks x and y within a policy π_T that builds a tower, it is necessary for π_T to repeatedly call π_{on} on the right sequence of blocks x and y.

In this work, we develop an extension of the language of policies and sketches that accommodates reuse. In the resulting framework, policies and sketches may call other policies and sketches while passing them arguments. For example, one can learn a policy π_T for building towers of blocks that uses a policy π_{on} for putting one block on top of another, and then reuse π_T for learning a policy that builds any configuration of blocks.

The language extensions involved are basically three: *memory states*, as in finite state controllers for sequencing behaviors, *indexical features*, whose values are a function of the state and a number of *internal registers* that can be loaded with objects, and *modules,* that wrap policies and sketches so that they can be reused by other policies or sketches. In this paper, we do not address the *learning* problem, but the *representation* problem, as in order to learn these policies bottom up, one must be able to represent them.

The paper is structured as follows. The next two sections cover related work and review planning and sketches. Then, two sections introduce extended policies and sketches, and their semantics, and the following section introduces modules, call rules, and action calls. The last section contains a discussion with conclusions and directions for future work.

Related Work

General policies. The paper builds on notions of width, generalized rule-based policies and sketches of bounded width, and methods for learning them (Lipovetzky and Geffner 2012; Bonet and Geffner 2018, 2023; Francès, Bonet, and Geffner 2021; Drexler, Seipp, and Geffner 2022, 2023). The problem of representing and learning general policies has a

[1]For each predicate p appearing in the goal, a new predicate p_G is introduced with the same arity as p. The atom $p(c)$ in the state s means that $p(c)$ is true in s, while $p_G(c)$ in s means that $p(c)$ must be true in the goal.

long history (Khardon 1999; Martín and Geffner 2004; Fern, Yoon, and Givan 2006), and general plans have also been represented in logic (Srivastava, Immerman, and Zilberstein 2011; Illanes and McIlraith 2019), and neural nets (Groshev et al. 2018; Toyer et al. 2020; Bueno et al. 2019; Rivlin, Hazan, and Karpas 2020; Garg, Bajpai, and Mausam 2020; Ståhlberg, Bonet, and Geffner 2022a,b, 2023).

Planning programs and inductive programming. Planning programs have been proposed as a language for representing and learning general policies that make use of a number of programming language constructs (Aguas, Celorrio, and Jonsson 2016; Segovia-Aguas, Jiménez, and Jonsson 2019, 2021). However, without a rich feature language for talking about goals, states, and their relation, planning programs are basically limited to families of tasks where the goal is fixed; like "putting all blocks on the table", but not for "building a given tower".[2] The problem of program reuse, which is related to the problem of policy reuse, has been considered in program synthesis and inductive programming (Dumancic, Guns, and Cropper 2021; Ellis et al. 2023).

Deictic representations. The use of registers to store objects in the extended language of policies and sketches is closely related to the use of indices and visual markers in deictic or indexical representations (Chapman 1989; Agre and Chapman 1990; Ballard et al. 1996). The computational value of such representations, however, has not been clear (Finney et al. 2013). In our setting, indices (registers) make policies (and sketches) parametric and more expressive, as (sub)policies can be reused by setting and resetting the values of registers as needed.

Hierarchical RL. Hierarchical structures have been used in RL in the form of options (Sutton, Precup, and Singh 1999), hierarchies of machines (Parr and Russell 1997) and MaxQ hierarchies (Dietterich 2000), and a vast literature has explored methods for learning hierarchical policies (McGovern and Barto 2001; Machado, Bellemare, and Bowling 2017), in some cases, representing goals and subgoals as part of an extended state (Kulkarni et al. 2016; Hafner et al. 2022). In almost all cases, however, policies and subpolicies are learned jointly, not bottom up, and the information that is conveyed by parameters, if they exist, is limited.

Planning Problems and Sketches

A **planning problem** refers to a classical planning problem $P = \langle D, I \rangle$ where D is the planning domain and I contains information about the instance; namely, the objects in the instance, the initial situation, and the goal. A class $\mathcal{Q}$ of planning problems is a set of instances over a common domain D. A **sketch** for a class of problems $\mathcal{Q}$ is a set of **rules** of the form $C \mapsto E$ based on **features** over the domain D which can be Boolean or numerical (i.e. non-negative integer valued) (Bonet and Geffner 2023). The condition C is a conjunction of expressions like p and $\neg p$ for Boolean features p, and $n > 0$ and $n = 0$ for numerical features n, while the effect E is a conjunction of expressions like p, $\neg p$, and $p?$, and $n\downarrow$, $n\uparrow$ and $n?$ for Boolean and numerical features p and n, respectively. A state pair (s, s') over an instance P in $\mathcal{Q}$ is **compatible** with a rule r if the state s satisfies the condition C, and the change of feature values from s to s' is consistent with E. This is written as $s' \prec_r s$ and $s' \prec_R s$ if R is a set of rules that contains r.

[2]The goal is fixed in a class $\mathcal{Q}$ of planning problems when for any P in $\mathcal{Q}$, the goal G of P is determined by its initial state.

Algorithm 1: SIW_R: sketch R is used to decompose problem into subproblems, each solved with the IW algorithm.

```
1: **Input:** Sketch $R$ over features $\Phi$ that induces relation $\prec_R$
2: **Input:** Planning problem $P$ with initial state $s_0$ on which the
   features in $\Phi$ are well defined
3: $s \leftarrow s_0$
4: **while** $s$ is not a goal state of $P$ **do**
5:     Run IW search from $s$ to find goal state $s'$ of $P$, or state
           $s'$ such that $s' \prec_R s$
6:     **if** $s'$ is not found, **return** FAILURE
7:     $s \leftarrow s'$
8: **return** path from $s_0$ to the goal state $s$
```

A sketch R for a class $\mathcal{Q}$ splits the problems P in $\mathcal{Q}$ into **subproblems** $P[s]$ that are like P but with initial state s (where s is a reachable state in P), and goal states s' that are either goal states of P, or states s' such that $s' \prec_R s$. The algorithm SIW_R shown in Alg 1 uses this problem decomposition to solve problems P in $\mathcal{Q}$ by solving subproblems $P[s]$ via the IW algorithm (Lipovetzky and Geffner 2012). If the sketch has bounded serialized width over $\mathcal{Q}$ and is terminating, SIW_R solves any problem P in $\mathcal{Q}$ in polynomial time (Bonet and Geffner 2023; Srivastava et al. 2011).

Serialized Width, Acyclicity, and Termination

The **width** of a planning problem P provides a complexity measure for finding an optimal plan for P. If P has N ground atoms and its width is bounded by k, written $w(P) \leq k$, an optimal plan for P can be found in $\mathcal{O}(N^{2k-1})$ time and $\mathcal{O}(N^k)$ space by running the algorithm IW(k), which is a simple breadth-first search where newly generated states are pruned if they do not make a tuple (set) of k or less atoms true for the first time in the search (Lipovetzky and Geffner 2012). If the width of P is bounded but its value is unknown, a plan (not necessarily optimal) can be found by running the algorithm IW with the same complexity bounds. IW calls IW(i) iteratively with $i = 0, \ldots, N$, until P is solved (Lipovetzky and Geffner 2012). If P has no solution, its width is defined as $w(P) \doteq \infty$, and if it has a plan of length one, as $w(P) \doteq 0$. The width notion extends to classes $\mathcal{Q}$ of problems: $w(\mathcal{Q}) \leq k$ iff $w(P) \leq k$ for each problem P in $\mathcal{Q}$. If $w(\mathcal{Q}) \leq k$ holds for class $\mathcal{Q}$, then any problem P in $\mathcal{Q}$ can be solved in **polynomial time** as k is independent of the size of P.

A sketch R has **serialized width** bounded by k on a class $\mathcal{Q}$, denoted as $w_R(\mathcal{Q}) \leq k$, if for any P in $\mathcal{Q}$, the possible subproblems $P[s]$ have width bounded by k. In such a case, every subproblem $P[s]$ that arises when running the SIW_R algorithm on P can be solved in **polynomial time.**

Finally, a sketch R is **acyclic** in P if there is no state sequence $s_0, s_1, \ldots, s_n$ in P such that $s_{i+1} \prec_R s_i$, for $0 \leq i < n$, and $s_n = s_0$, and it is acyclic in $\mathcal{Q}$ if it is acyclic in each problem P in $\mathcal{Q}$. The SIEVE algorithm (Srivastava et al. 2011) can check whether a sketch R is **terminating**, and hence acyclic, by just considering the rules in R and the graph that they define. For a terminating sketch R with a bounded serialized width over $\mathcal{Q}$, SIW_R is guaranteed to find a solution to any problem P in $\mathcal{Q}$ in **polynomial time** (Bonet and Geffner 2023).

Example 1. A sketch of width 0 (i.e., a policy) for achieving the atom $on(x, y)$ for two blocks x and y, on any Blocksworld instance, can be defined with the numerical feature n that counts the number of blocks above x, y, or both, and the Boolean features *On* that represents whether the goal holds (i.e., x on y), H that represents whether a block is being held, and H_x that represents whether the block x is being held. The set of features is $\Phi = \{On, H, H_x, n\}$, and the rules are:

$$\begin{aligned}
r_0 &= \{\neg On, n > 0, \neg H, \neg H_x\} \mapsto \{n\downarrow, H, H_x?\}\\
r_1 &= \{\neg On, n > 0, H\} \mapsto \{\neg H, \neg H_x\}\\
r_2 &= \{\neg On, n = 0, \neg H, \neg H_x\} \mapsto \{H, H_x\}\\
r_3 &= \{\neg On, n = 0, H, H_x\} \mapsto \{On, n\uparrow, \neg H, \neg H_x\}
\end{aligned}$$

Rule r_0 says that in states where no block is being held and $n > 0$, picking a block above x or y is good, as it would decrement n and make H true. Rule r_1 says that in states where a block is being held and $n > 0$, putting the block away from x or y is good, as the resulting state transition would not change the value of n. Last, rule r_2 says that in states where no block is held and $n = 0$, picking up x is good, and r_3, that in states where block x is held and $n = 0$, putting x on y is good, as it results in *On* being true, while also increasing n and making H and H_x false.

A sketch of width 2 for $\mathcal{Q}_{on}$ is obtained instead by replacing r_0 and r_1 with the rule $\{\neg On, n > 0\} \mapsto \{n\downarrow, \neg H\}$, and r_2 and r_3 with the rule $\{\neg On, n = 0\} \mapsto \{On, n\uparrow, \neg H\}$. ■

Extended Sketches

The first two extensions of sketches are introduced next.

Finite Memory

The first language extension adds memory in the form of a finite number of *memory states* m:

Definition 1 (Sketches with Memory)**.** *A sketch with finite memory is a tuple $\langle M, \Phi, m_0, R\rangle$ where M is a finite set of memory states, Φ is a set of features, $m_0 \in M$ is the initial memory state, and R is a set of rules extended with memory states. Such rules have the form $(m, C) \mapsto (E, m')$ where $C \mapsto E$ is a standard sketch rule, and m and m' are memory states in M.*

When the current memory state is m, only rules of the form $(m, C) \mapsto (E, m')$ apply. If s and m are the current state and memory, respectively, and s' is a state reachable from s such that the pair (s, s') is compatible with the rule $C \mapsto E$, then moving to state s' and setting the memory to m' is compatible with the extended rule $(m, C) \mapsto (E, m')$. In displayed listings, rules $(m, c) \mapsto (E, m')$ are written as $m \,\|\, C \mapsto E \,\|\, m'$ to improve readability.

Example 2. Liu et al. (2023) describe a simple policy for solving the class $\mathcal{Q}_{Hanoi}$ of Towers-of-Hanoi instances with 3 pegs, where a tower in the first peg is to be moved to the third peg. In this policy, actions alternate between moving the smallest top disk and moving the second smallest top disk. When moving the smallest top disk, it is always moved to the "left", towards the first peg, except when it is in the first peg that is then moved to the third peg. In the alternate step, the second smallest top disk is moved on top of the third smallest top disk, as no other choice is available.

The movements can be expressed in the language of sketches with three Boolean features $p_{i,j}$, $1 \leq i < j \leq 3$, that are true if the top disk at peg i is smaller than the top disk at peg j. For example, the smallest disk is at the first (resp. third) peg iff $p_{1,2} \wedge p_{1,3}$ (resp. $\neg p_{1,3} \wedge \neg p_{2,3}$) holds.

The alternation of actions is obtained with two memory states m_0 and m_1 of which m_0 is the initial memory state. The rules implementing the policy are then:

% Movements of the smallest disk

$$\begin{aligned}
r_0 &= m_0 \,\|\, \{p_{1,2}, p_{1,3}\} \mapsto \{p_{1,2}?, \neg p_{1,3}, \neg p_{2,3}\} \,\|\, m_1\\
r_1 &= m_0 \,\|\, \{\neg p_{1,2}, p_{2,3}\} \mapsto \{p_{1,2}, p_{1,3}, p_{2,3}?\} \,\|\, m_1\\
r_2 &= m_0 \,\|\, \{\neg p_{1,3}, \neg p_{2,3}\} \mapsto \{\neg p_{1,2}, p_{1,3}?, p_{2,3}\} \,\|\, m_1
\end{aligned}$$

% Movements of the second smallest disk

$$\begin{aligned}
r_3 &= m_1 \,\|\, \{p_{1,2}, p_{1,3}, p_{2,3}\} \mapsto \{\neg p_{2,3}\} \,\|\, m_0\\
r_4 &= m_1 \,\|\, \{p_{1,2}, p_{1,3}, \neg p_{2,3}\} \mapsto \{p_{2,3}\} \,\|\, m_0\\
r_5 &= m_1 \,\|\, \{\neg p_{1,2}, p_{1,3}, p_{2,3}\} \mapsto \{\neg p_{1,3}\} \,\|\, m_0\\
r_6 &= m_1 \,\|\, \{\neg p_{1,2}, \neg p_{1,3}, p_{2,3}\} \mapsto \{p_{1,3}\} \,\|\, m_0\\
r_7 &= m_1 \,\|\, \{p_{1,2}, \neg p_{1,3}, \neg p_{2,3}\} \mapsto \{\neg p_{1,2}\} \,\|\, m_0\\
r_8 &= m_1 \,\|\, \{\neg p_{1,2}, \neg p_{1,3}, \neg p_{2,3}\} \mapsto \{p_{1,2}\} \,\|\, m_0
\end{aligned}$$

The policy corresponds to the sketch with memory $\langle M, \Phi, m_0, R\rangle$ where $M = \{m_0, m_1\}$, $\Phi = \{p_{i,j} : 1 \leq i < j \leq 3\}$, and R is the set of above rules. Notice that this policy is guaranteed to solve only instances of Towers-of-Hanoi with 3 pegs where the initial state contains a single tower in the first peg that is to be moved to the third peg. ■

Registers, Indexicals, Concepts, and Roles

The second extension introduces internal memory in the form of *registers*. Registers store objects, which can be referred to in features that become indexical or parametric, as their value changes when the object in the register changes; e.g., the number of blocks above the block in register zero. The objects that can be placed into the registers $\mathfrak{R} = \{\mathfrak{r}_0, \mathfrak{r}_1, \ldots\}$ are selected by two new classes of features called *concepts* and *roles* that denote sets of objects and set of object pairs, respectively (Baader et al. 2003).

Concept features or simply concepts are denoted in sans-serif font such as 'C', and in a state s, they denote the set of objects in the problem P that satisfy the unary predicate 'C'. Role features or simply roles are also denoted in sans-serif font such as 'R', and in a state s, they denote the set of object pairs in the problem P that satisfy the binary relation 'R'. Concept and role features are also used as numerical features, e.g., as conditions '$\mathsf{C} > 0$' or effects '$\mathsf{R}\downarrow$', with the understanding that the corresponding numerical feature is given by the cardinality of the concept C or role R in the state; namely, the number of objects or object pairs in their denotation.

While a plain feature is a function of the problem state, an indexical or parametric feature is a function of the problem state and the value of the registers; like "the distance of the agent to the object stored in $\mathfrak{r}_0$". When the value of a register $\mathfrak{r}$ changes, the denotation of indexical features that depend on the value of the register may change as well. We denote by $\Phi(\mathfrak{r})$ the subset of features in Φ that refer to (i.e., depend on the value of) register $\mathfrak{r}$. The set of features Φ is assumed to contain a (parametric) concept for each register $\mathfrak{r}$, whose denotation is the singleton that contains the object in $\mathfrak{r}$, and that is also denoted by $\mathfrak{r}$.

The extended sketch language provides *load effects* of the form $\mathit{Load}(\mathsf{C},\mathfrak{r})$ for updating the value of registers for a concept C and register $\mathfrak{r}$; an expression that indicates that the content of the register $\mathfrak{r}$ is to be set to *any object* in the (current) denotation of C, a choice that is *non-deterministic*. Loading an object into register $\mathfrak{r}$ can be thought as placing the *marker* $\mathfrak{r}$ on the object. A rule with effect $\mathit{Load}(\mathsf{C},\mathfrak{r})$ has the condition $\mathsf{C}>0$ to ensure that C contains some object. Likewise, since a load may change the denotation of features, the effect of a load rule on register $\mathfrak{r}$ is assumed to contain also the extra effects $\phi?$ for the features ϕ in $\Phi(\mathfrak{r})$, and no other effects. Formally,

Definition 2 (Extended Sketch Rules)**.** *An **extended rule** over the features Φ and memory M has the form $(m,C)\mapsto(E,m')$ where m and m' are memory states, C is a Boolean condition on the features, and either E expresses changes in the feature values, as usual, or E contains one load effect of the form $\mathit{Load}(\mathsf{C},\mathfrak{r})$ for some concept C and register $\mathfrak{r}$. If the latter case, E must also contain uncertain effects $\phi?$ on the features ϕ in $\Phi(\mathfrak{r})$, but no other effects.*

Rules with a load effect are called *internal rules*, as they capture changes in the internal memory, while the other rules are called *external.* For simplicity, it is assumed, that each load rule contains a single load effect, and that internal and external rules apply in different memory states, from now called *internal* and *external* memory states. For convenience, rules of the form $(m,C)\mapsto(true,m')$, abbreviated as $(m,C)\mapsto(\{\},m')$, are allowed to enable a change from memory state m to memory state m' under the condition C. Extended sketches are comprised of extended sketch rules:

Definition 3 (Extended Sketch)**.** *An **extended sketch** is a tuple $\langle M,\mathfrak{R},\Phi,m_0,R\rangle$ where M and $\mathfrak{R}$ are finite sets of memory states and registers, respectively, Φ is a set of features, $m_0\in M$ is the initial memory state, and R is a set of extended Φ-rules over memory M. If m is a memory state, $R(m)$ denote the subset of rules in R of form $(m,C)\mapsto(E,m')$.*

Extended sketches are also called indexical sketches, and those of width 0 are called extended or indexical policies.

Example 3. A different, indexical, policy π^*_{on} for the class $\mathcal{Q}_{on}$ of Blocksworld problems considered in Example 1, where the goal is to achieve the atom $on(x,y)$, is expressed as the sketch $\langle M,\mathsf{R},\Phi,m_0,R\rangle$ where $M=\{m_i:0\leq i\leq 8\}$ has 9 memory states, and $\mathsf{R}=\{\mathfrak{r}_0,\mathfrak{r}_1\}$ has two registers. The set of features is $\Phi=\{\mathit{On},H,H_x,A,\mathsf{N},\mathsf{T}_0,\mathsf{T}_1\}$ where *On*, H and H_x are as in Example 1, A is true if the block in $\mathfrak{r}_1$ is above x or y, N is the subset of blocks in $\{x,y\}$ that are *not clear,* and the concept T_0 (resp. T_1) contains the block, if any, that is directly above the block in $\mathfrak{r}_0$ (resp. $\mathfrak{r}_1$). The set R has the following 14 rules:

% Internal rules (update registers and internal memory)
% Initial border case when holding some block: apply rule r_8
$r_0\ =\ m_0\,\|\,\{H\}\mapsto\{\}\,\|\,m_4$

% Blocks x and y already clear: apply rule r_{12}
$r_1\ =\ m_0\,\|\,\{\neg H,\mathsf{N}=0\}\mapsto\{\}\,\|\,m_7$

% Place $\mathfrak{r}_0$ on some block in N, and $\mathfrak{r}_1$ on topmost above $\mathfrak{r}_0$
$r_2\ =\ m_0\,\|\,\{\neg H,\mathsf{N}>0\}\mapsto\{\mathit{Load}(\mathsf{N},\mathfrak{r}_0),\mathsf{T}_0?\}\,\|\,m_1$
$r_3\ =\ m_1\,\|\,\{\mathsf{T}_0>0\}\mapsto\{\mathit{Load}(\mathsf{T}_0,\mathfrak{r}_1),\mathsf{T}_1?,A?\}\,\|\,m_2$
$r_4\ =\ m_2\,\|\,\{\mathsf{T}_1>0\}\mapsto\{\mathit{Load}(\mathsf{T}_1,\mathfrak{r}_1),\mathsf{T}_1?,A?\}\,\|\,m_2$
$r_5\ =\ m_2\,\|\,\{\mathsf{T}_1=0\}\mapsto\{\}\,\|\,m_5$

% Iterate over next block above $\mathfrak{r}_0$, or next block in N
$r_6\ =\ m_3\,\|\,\{\mathsf{T}_0>0\}\mapsto\{\}\,\|\,m_1$
$r_7\ =\ m_3\,\|\,\{\mathsf{T}_0=0\}\mapsto\{\}\,\|\,m_0$

% External rules (involve state transitions)
% Handle initial border case: drop block being held
$r_8\ =\ m_4\,\|\,\{H\}\mapsto\{\neg H,\neg H_x,\mathit{On}?,A?,\mathsf{N}?,\mathsf{T}_0?,\mathsf{T}_1?\}\,\|\,m_0$

% Pick block $\mathfrak{r}_1$ (topmost above $\mathfrak{r}_0$) and put it away
$r_9\ =\ m_5\,\|\,\{\neg H,A\}\mapsto\{H,\neg A,\mathsf{N}?,\mathsf{T}_0?\}\,\|\,m_6$
$r_{10}\ =\ m_6\,\|\,\{H,\neg A\}\mapsto\{\neg H\}\,\|\,m_3$
$r_{11}\ =\ m_6\,\|\,\{H,\neg A\}\mapsto\{\neg H,\mathsf{N}\!\downarrow\}\,\|\,m_3$

% Pick up block x and place on block y
$r_{12}\ =\ m_7\,\|\,\{\neg H_x\}\mapsto\{H,H_x,\mathsf{N}?\}\,\|\,m_8$
$r_{13}\ =\ m_8\,\|\,\{H_x,\neg\mathit{On}\}\mapsto\{\neg H,\neg H_x,\mathit{On}\}\,\|\,m_8$

The policy π^*_{on} is *indexical* (Agre and Chapman 1990; Ballard et al. 1996), as it puts the *"mark"* $\mathfrak{r}_0$ on a block in N (rule r_2), while a second *mark* $\mathfrak{r}_1$ is moved up from the block on $\mathfrak{r}_0$ to the topmost block above $\mathfrak{r}_0$ (rules r_3–r_5). The block marked as $\mathfrak{r}_1$ is then picked and put away (rules r_9–r_{11}), and the process repeats (loop on rules $r_6,r_3,r_4^+,r_5,r_9,r_{10},r_6$) until the block in $\mathfrak{r}_0$ becomes clear. Then, another block in N, if any, is loaded into $\mathfrak{r}_0$ (rules r_7,r_2). Once x and y become clear (i.e., $\mathsf{N}=0$), the block x is picked and placed on y (rules r_1, r_{12} and r_{13}). The rule r_0 handles the special case when initially some block is being held.

The new policy, unlike the one in Example 1, does not use features for counting blocks. The features used are indeed computationally simpler as they refer to "markers" (i.e., registers that store or mark specific objects) that *fix the attention* on specific blocks, something which non-indexical features cannot do. Figure 1 illustrates the use of markers on a tower of 4 blocks where x is at the bottom. ■

Formal Semantics and Termination

Extended sketches are evaluated on planning states augmented with memory states and register values.

Definition 4 (Augmented states)**.** *An **augmented state** for a problem P given an extended sketch $\langle M,\mathfrak{R},\Phi,m_0,R\rangle$ is a tuple $\bar{s}=(s,m,\boldsymbol{v})$ where s is a reachable state in P, m is a memory state in M, and $\boldsymbol{v}$ is a vector of objects in $\mathit{Obj}(P)^{\mathfrak{R}}$ that tells the content of each register $\mathfrak{r}$, denoted as $\boldsymbol{v}[\mathfrak{r}]$.*

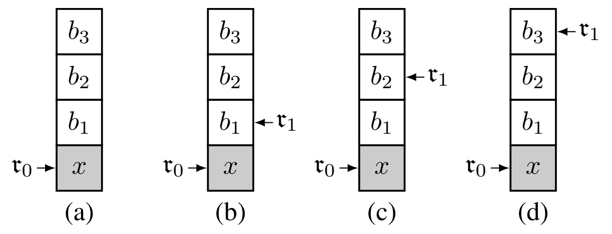

Figure 1: Policy π^*_{on} places markers $\mathfrak{r}_0$ and $\mathfrak{r}_1$ on blocks x and b_3, respectively, for an example tower with 4 blocks. (a) The rule r_2 puts the marker $\mathfrak{r}_0$ on the block $x \in \mathsf{N}$; side effect is $\mathsf{T}_0 = \{b_1\}$. (b) Rule r_3 initializes marker $\mathfrak{r}_1$ on block $b_1 \in \mathsf{T}_0$; side effect is $\mathsf{T}_1 = \{b_2\}$. (c) Rule r_4 moves marker $\mathfrak{r}_1$ one step above to block $b_2 \in \mathsf{T}_1$; side effect is $\mathsf{T}_1 = \{b_3\}$. (d) Another application of r_4 moves $\mathfrak{r}_1$ on block $b_3 \in \mathsf{T}_1$; side effect is $\mathsf{T}_1 = \emptyset$, and r_5 is the next rule to apply.

Features ϕ are evaluated over pairs $(s, \boldsymbol{v})$ made up of a state s and a value $\boldsymbol{v}$ for the registers. The value for ϕ at such a pair is denoted by $\phi(s, \boldsymbol{v})$.

Definition 5 (Compatible pairs)**.** *Let r be an **external** Φ-rule $(m, C) \mapsto (E, m')$, and let $\boldsymbol{v}$ be a valuation for the registers. A state s satisfies the condition C **given** $\boldsymbol{v}$ if the feature conditions in C are all true in $(s, \boldsymbol{v})$. A state pair (s, s') satisfies the effect E **given** $\boldsymbol{v}$ if the values for the features in Φ change from s to s' according to E; i.e., the following holds where p is a Boolean feature, n is a numerical, concept or role feature, and ϕ is any type of feature,*

1. *if p (resp. $\neg p$) is in E, $p(s', \boldsymbol{v}) = 1$ (resp. $p(s', \boldsymbol{v}) = 0$),*
2. *if $n\downarrow$ (resp. $n\uparrow$) is in E, $n(s, \boldsymbol{v}) > n(s', \boldsymbol{v})$ (resp. $n(s, \boldsymbol{v}) < n(s', \boldsymbol{v})$), and*
3. *if ϕ is not mentioned in E, $\phi(s, \boldsymbol{v}) = \phi(s', \boldsymbol{v})$.*

*The pair (s, s') is **compatible** with an external rule $r = (m, C) \mapsto (E, m')$ **given** $\boldsymbol{v}$, denoted as $s' \prec_{r/\boldsymbol{v}} s$, if given $\boldsymbol{v}$, s satisfies C and the pair satisfies E. The pair is compatible with a set of rules R given $\boldsymbol{v}$, denoted as $s' \prec_{R/\boldsymbol{v}} s$, if it is compatible with some external rule in R given $\boldsymbol{v}$.*

The search algorithm for extended sketches, called SIW^*_R and shown in Alg. 2, maintains the current memory state and register values, implements the internal rules, and performs IW searches to solve the subproblems that arise when the memory state are external.

Definition 6 (Subproblems)**.** *If $\bar{s} = (s, m, \boldsymbol{v})$ is an augmented state for P, where m is external memory, the subproblem $P[\bar{s}]$ induced by $\bar{s}$ in P is a planning problem which is like P but with initial state s, and goal states that are either goal states of P, or states s' such that $s' \prec_{r/\boldsymbol{v}} s$ for some rule r in $R(m)$.*

Internal rules do not generate classical subproblems and do not change the planning state s but they affect the memory state and the register values, and with that, the definition of the subproblems that follow. The notion of **reduction** captures how internal rules are processed:

Definition 7 (Reduction)**.** *Let $\langle M, \mathfrak{R}, \Phi, m_0, R\rangle$ be an extended sketch for a planning problem P, and let $(s, m, \boldsymbol{v})$ be an augmented state for P where m is an internal memory state in M. The pair $\langle (s, m, \boldsymbol{v}), (s, m', \boldsymbol{v}')\rangle$ is a **reduction step** if there is a rule r in R of form $(m, C) \mapsto (E, m')$ such that 1) s satisfies the condition C given $\boldsymbol{v}$, and 2) either E is a dummy true effect and $\boldsymbol{v}' = \boldsymbol{v}$, or E contains Load($\mathsf{C}, \mathfrak{r}$), and $\boldsymbol{v}'[\mathfrak{r}] \in \mathsf{C}(s, \boldsymbol{v})$. A sequence of reduction steps starting at $(s, m, \boldsymbol{v})$ and ending at $(s, m', \boldsymbol{v}')$ where m' is external is called a **reduction**, and it is denoted by $(s, m, \boldsymbol{v}) \to^* (s, m', \boldsymbol{v}')$. For convenience, if m is external, the triplet $(s, m, \boldsymbol{v})$ is assumed to reduce to itself, written $(s, m, \boldsymbol{v}) \to^* (s, m, \boldsymbol{v})$.*

An **initial augmented state** for problem P is of the form $(s_0, m, \boldsymbol{v})$ where s_0 is the initial state in P, and $(s_0, m_0, \boldsymbol{v}_0) \to^* (s_0, m, \boldsymbol{v})$ for the initial memory m_0 and some $\boldsymbol{v}_0$ in $\text{Obj}(P)^{\mathfrak{R}}$. There may be different initial augmented states for P that differ in their memory and/or the contents of the registers. Each such initial augmented state defines an **initial subproblem** $P[s_0, m, \boldsymbol{v}]$ (cf. Definition 6).

Definition 8 (Induced subproblems)**.** *Let $\langle M, \mathfrak{R}, \Phi, m_0, R\rangle$ be an extended sketch for a planning problem P with initial state s_0. Let us consider a subproblem $P[\bar{s}]$ for $\bar{s} = (s, m, \boldsymbol{v})$ where m is external, and let s' be a state reachable from s with $s' \prec_{r/\boldsymbol{v}} s$ for some rule r in $R(m)$. Then,*

1. *subproblem $P[s', m', \boldsymbol{v}]$ is **induced** by subproblem $P[\bar{s}]$ if $r = (m, C) \mapsto (E, m')$ and m' is external memory.*
2. *Subproblem $P[s', m'', \boldsymbol{v}']$ is **induced** by subproblem $P[\bar{s}]$ if $r = (m, C) \mapsto (E, m')$, m' is internal memory, and $(s', m', \boldsymbol{v}) \to^* (s', m'', \boldsymbol{v}')$.*

The collection P° of induced subproblems is the smallest set such that 1) P° contains all the initial subproblems, and 2) $P[s', m', \boldsymbol{v}']$ is in P° if $P[s, m, \boldsymbol{v}]$ is in P° and the first subproblem is induced by the second.

In this definition, subproblems $P[s, m, v]$ where m is internal are "jumped over" so that the subproblems that make it into P° all have memory states m that are external, and hence represent classical planning problems and not bookkeeping operations. The extended sketch R is said to be **reducible** in P if for any reachable augmented state $(s, m, \boldsymbol{v})$ in P where m is an internal state, there is an augmented state $(s, m', \boldsymbol{v}')$ such that $(s, m, \boldsymbol{v}) \to^* (s, m', \boldsymbol{v}')$. A non-reducible sketch is one in which the executions can cycle or get stuck while performing internal memory operations.

Definition 9 (Sketch width)**.** *The **width** of a reducible sketch R over a planning problem P is bounded by non-negative integer k, denoted by $w_R(P) \leq k$, if the width of each subproblem in P° is bounded by k. The **width** of a reducible sketch R over a class of problems $\mathcal{Q}$ is bounded by k, denoted by $w_R(\mathcal{Q}) \leq k$, if $w_R(P) \leq k$ for each P in $\mathcal{Q}$.*

The **width** is zero for the indexical sketch π^*_{on} in Example 3, as the sketch represents a policy where each subproblem is solved in a single step.

Termination for Extended Sketches

Termination is a key property for sketches that guarantees acyclicity, and that only a polynomial number of subproblems may appear when solving *any* problem P where the

Algorithm 2: SIW$^*_{\mathrm{R}}$ uses the extended sketch R to decompose problem P into subproblems that are solved with IW. Completeness of SIW$^*_{\mathrm{R}}$ is captured in Theorem 10.

```
1:  Input: Extended sketch $\langle M, \mathfrak{R}, \Phi, m_0, R\rangle$
2:  Input: Planning problem $P$ with initial state $s_0$ on which the
    features in $\Phi$ are well defined
3:  $\bar{s} \leftarrow (s_0, m_0, \boldsymbol{v})$ for some $\boldsymbol{v} \in \text{Obj}(P)^{\mathfrak{R}}$
4:  while $s$ in $\bar{s} = (s, m, \boldsymbol{v})$ is not a goal state of $P$
5:    if $m$ is internal memory
6:      Find rule $r = (m, C) \mapsto (E, m')$ with $s, \boldsymbol{v} \vDash C$
7:      if $r$ is not found, return FAILURE          % Irreducible
8:      if $Load(\mathsf{C}, \mathfrak{r})$ in $E$, $\boldsymbol{v}[\mathfrak{r}] \leftarrow o$ for some $o \in \mathsf{C}(s, \boldsymbol{v})$
9:      $\bar{s} \leftarrow (s, m', \boldsymbol{v})$
10:   else                       % $m$ is external, solve subproblem
11:     Run IW search from $s$ to find goal state $s'$ of $P$, or
          state $s'$ such that $s' \prec_{r/\boldsymbol{v}} s$ for some (external)
          rule $r = (m, C) \mapsto (E, m')$ in $R$
12:     if no such state is found, return FAILURE
13:     $\bar{s} \leftarrow (s', m', \boldsymbol{v})$
14: return path from $s_0$ to the goal state $s$
```

features are well defined (Bonet and Geffner 2023). Termination can be tested in polynomial time by a suitable adaptation of the SIEVE algorithm (Srivastava et al. 2011; Bonet and Geffner 2023). If the extended sketch is reducible, terminating, and has bounded serialized width over a class $\mathcal{Q}$, any problem P in $\mathcal{Q}$ can be solved in polynomial time with the procedure SIW$^*_{\mathrm{R}}$ shown in Alg. 2.

Theorem 10 (Termination)**.** *If the extended sketch R is* ***reducible****,* ***terminating****, and has a* ***serialized width*** *bounded by k over the class of problems $\mathcal{Q}$, then SIW$^*_{\mathrm{R}}$ finds plans for any problem P in $\mathcal{Q}$ in* ***polynomial time.***

Reusable Modules

Policies and sketches are wrapped into *modules* to enable calls among them. A module is a named tuple $\langle \text{args}, Z, M, \mathfrak{R}, \Phi, m_0, R\rangle$ where $\text{args} = \langle x_1, x_2, \ldots, x_n\rangle$ is a tuple of arguments, each one being either a *static concept or role*, Z and Φ are sets of features, M is a set of memory states, $\mathfrak{R}$ is a set of registers, $m_0 \in M$ is the initial memory state, and R is a set of rules. The features in Φ are the ones mentioned in the sketch rules in R, and their denotation may depend on the value of the arguments and the features in Z, and the contents of the registers. The sketch R in a module may also contain two new types of *external* rules, *call rules* and *do rules*. The first type permits to call other modules; the second type permits the *direct execution of ground actions of the planning problem*. The value for the arguments in either case is given by the features in Φ or Z, and the arguments of the module; the difference between Φ and Z is that the rules in R must track the changes for the features in Φ, while Z can be used to define features in Φ, appear in conditions of rules, and provide values to arguments in call and do rules. If the name of the module is `name`, a *call* to the module is expressed as $\texttt{name}(x_1, x_2, \ldots, x_n)$ where $x_1, \ldots, x_n$ are the module arguments.

Call and do *rules* are of the form $(m, C) \mapsto (\texttt{name}(v_1, v_2, \ldots, v_n), m')$ where m is internal memory, C is a condition, `name` is a module or action schema name, and each value v_i is of an appropriate type: for *do rules*, v_i must be a concept, while for *call rules*, v_i can be either a concept or a role. The idea is that if a *call rule* is used, the sketch associated with the module `name` is executed until no rules are applicable, and the control is then returned back to the caller at the memory state m'. For *do rules*, an applicable ground action of the form $\texttt{name}(o_1, o_2, \ldots, o_n)$, where the object o_i belongs to the denotation of concept v_i, must be applied at the current state to make a transition to a successor state, and control is then returned back to the caller at the memory state m'.

The execution model for handling modules involves a stack as described below. Modules call each other by passing arguments but do not get back any values. The "side effects" of a module are in the problem state s that must be driven eventually to a goal state.

Example 4. Module $\texttt{on}(\mathsf{X}, \mathsf{Y})$ for a policy for the class $\mathcal{Q}_{on}$ that implements a variant of the policy π^*_{on} in Example 3. Indeed, *by directly executing ground actions*, the policy uses simpler features and can be executed model-free: there is no need to generate the possible successors for choosing the ground action to do. This is directly encoded in the policy. For example, the indexical Boolean feature A is no longer needed because the block being held is put away on the table with a `Putdown` action.

The module $\texttt{on}(\mathsf{X}, \mathsf{Y})$ has as parameters the concepts X and Y that are assumed to be singletons containing the blocks x and y, respectively. The module uses *do rules* to avoid references to complex concepts; it only uses the concepts B for all blocks, N for the blocks in $\mathsf{X} \cup \mathsf{Y}$ that are not clear, and the indexical concept T_0 (resp. T_1) for the block directly above $\mathfrak{r}_0$ (resp. $\mathfrak{r}_1$), if any. The other feature is the Boolean T_{X} that is true iff X is on the table. The module uses 9 memory states, two registers, and the following rules:

% Internal rules (update registers and internal memory)

$r_0 \;=\; m_0 \,\|\, \{H\} \mapsto \{\} \,\|\, m_4$
$r_1 \;=\; m_0 \,\|\, \{\neg H, \mathsf{N}\!=\!0\} \mapsto \{\} \,\|\, m_7$
$r_2 \;=\; m_0 \,\|\, \{\neg H, \mathsf{N}\!>\!0\} \mapsto \{Load(\mathsf{N}, \mathfrak{r}_0), \mathsf{T}_0?\} \,\|\, m_1$
$r_3 \;=\; m_1 \,\|\, \{\mathsf{T}_0\!>\!0\} \mapsto \{Load(\mathsf{T}_0, \mathfrak{r}_1), \mathsf{T}_1?\} \,\|\, m_2$
$r_4 \;=\; m_2 \,\|\, \{\mathsf{T}_1\!>\!0\} \mapsto \{Load(\mathsf{T}_1, \mathfrak{r}_1), \mathsf{T}_1?\} \,\|\, m_2$
$r_5 \;=\; m_2 \,\|\, \{\mathsf{T}_1\!=\!0\} \mapsto \{\} \,\|\, m_5$
$r_6 \;=\; m_3 \,\|\, \{\mathsf{T}_0\!>\!0\} \mapsto \{\} \,\|\, m_1$
$r_7 \;=\; m_3 \,\|\, \{\mathsf{T}_0\!=\!0\} \mapsto \{\} \,\|\, m_0$

% External rules (involve state transitions)

$r_8 \;=\; m_4 \,\|\, \{\} \mapsto \texttt{Putdown}(\mathsf{B}) \,\|\, m_0$
$r_9 \;=\; m_5 \,\|\, \{\} \mapsto \texttt{Unstack}(\mathfrak{r}_1, \mathsf{B}) \,\|\, m_6$
$r_{10} \;=\; m_6 \,\|\, \{\} \mapsto \texttt{Putdown}(\mathfrak{r}_1) \,\|\, m_3$
$r_{11} \;=\; m_7 \,\|\, \{T_{\mathsf{X}}\} \mapsto \texttt{Pickup}(\mathsf{X}) \,\|\, m_8$
$r_{12} \;=\; m_7 \,\|\, \{\neg T_{\mathsf{X}}\} \mapsto \texttt{Unstack}(\mathsf{X}, \mathsf{B}) \,\|\, m_8$
$r_{13} \;=\; m_8 \,\|\, \{\} \mapsto \texttt{Stack}(\mathsf{X}, \mathsf{Y}) \,\|\, m_8$

Rules r_0–r_7 are like the ones for the indexical policy π^*_{on}. The external rules, however, are do-rules that apply ground actions to remove blocks above $\mathfrak{r}_0$, and to pick X and put it on Y. The Boolean T_{X} is used to decide whether to use a `Pickup` or `Unstack` action to grab X. ■

Example 5. The module $\texttt{tower}(\mathsf{O},\mathsf{X})$ is aimed at the class $\mathcal{Q}_{tower}$ of problems where blocks are to be stacked into a *single tower* resting on the table. The goal of such a problem is described by a conjunction of atoms $\wedge_{i=1}^{k} on(x_i, x_{i-1})$ and $ontable(x_0)$. The module is the tuple $\langle\langle \mathsf{O},\mathsf{X}\rangle, Z, M, \mathfrak{R}, \Phi, m_0, R\rangle$ where O is a role argument whose denotation contains the pairs $\{(x_i, x_{i-1}) : i = 1, ..., k\}$, and X is a concept argument that denotes the lowest block in the target tower that is misplaced.[3] The other elements in the module are $Z = \emptyset$, $M = \{m_0, m_1, \ldots, m_3\}$, $\mathfrak{R} = \{\mathfrak{r}_0\}$, and a set of features $\Phi = \{\mathsf{M}, \mathsf{W}\}$ where M is the indexical concept that contains the block to be placed above the block in $\mathfrak{r}_0$ according to O, if any, and, W is the indexical concept that contains the block directly below the block in $\mathfrak{r}_0$, if any, also according to the target tower O. The rules are:

% *Module* $\texttt{tower}(\mathsf{O},\mathsf{X})$
$r_0 \;=\; m_0 \,\|\, \{\mathsf{X} > 0\} \mapsto \{Load(\mathsf{X}, \mathfrak{r}_0), \mathsf{M}?, \mathsf{W}?\} \,\|\, m_1$
$r_1 \;=\; m_1 \,\|\, \{\mathsf{W} = 0\} \mapsto \texttt{on-table}(\mathfrak{r}_0) \,\|\, m_2$
$r_2 \;=\; m_1 \,\|\, \{\mathsf{W} > 0\} \mapsto \texttt{on}(\mathfrak{r}_0, \mathsf{W}) \,\|\, m_2$
$r_3 \;=\; m_2 \,\|\, \{\mathsf{M} > 0\} \mapsto \texttt{tower}(\mathsf{O}, \mathsf{M}) \,\|\, m_3$

The rule r_0 puts the lowest misplaced block in $\mathfrak{r}_0$, r_1 calls the module $\texttt{on-table}(\mathfrak{r}_0)$ to place $\mathfrak{r}_0$ on the table, r_2 calls the module $\texttt{on}$ to place $\mathfrak{r}_0$ on W, and r_3 **recursively calls** the module with the block that is supposed to be on $\mathfrak{r}_0$. The module $\texttt{on-table}(\mathsf{X})$, that puts the block in the singleton X on the table, is not spelled out. ■

Example 6. The module $\texttt{blocks}(\mathsf{O})$ solves arbitrary instances of Blocksworld. It works by calling the module $\texttt{tower}(\mathsf{O},\mathsf{X})$ with the parameter X being the singleton that contains the *lowest misplaced block* in one of the current towers. The module takes a single role argument O whose denotation encodes the pairs (x, y) corresponding to the goal atoms $on(x, y)$ as in Example 5. The module is the tuple $\langle\langle \mathsf{O}\rangle, Z, M, \mathfrak{R}, \Phi, m_0, R\rangle$ where $Z = \emptyset$, $M = \{m_0, m_1\}$, $\mathfrak{R} = \{r_0\}$, and $\Phi = \{\mathsf{L}\}$, where L contains the lowest misplaced blocks according to the pairs in O. The set R contains only 2 rules:

% *Module* $\texttt{blocks}(\mathsf{O})$
$r_0 \;=\; m_0 \,\|\, \{\mathsf{L} > 0\} \mapsto \{Load(\mathsf{L}, \mathfrak{r}_0)\} \,\|\, m_1$
$r_1 \;=\; m_1 \,\|\, \{\} \mapsto \texttt{tower}(\mathsf{O}, \mathfrak{r}_0) \,\|\, m_0$

The concept L is a high-complexity concept, meaning that computing its denotation at the current state is non-trivial. However, L can be removed as follows. First, a new marker $\mathfrak{r}$ is put on a block x such that the atoms $on_G(x, y)$ and $on(x, y')$, with $y \neq y'$, hold at the current state s; i.e., $\mathfrak{r}$ is put on a misplaced block x. Second, move the marker $\mathfrak{r}$ to y if y is also misplaced, and keep moving it down until the marked block is on a well-placed block. At this moment, the marked block is a lowest misplaced block. ■

Execution Model: SIW$_\mathrm{M}$

The execution model for modules is captured by the SIW$_\mathrm{M}$ algorithm in Alg. 3 which uses a stack and a caller/callee protocol, as it is standard in programming languages. It assumes a collection $\{\texttt{mod}_0, \texttt{mod}_1, \ldots, \texttt{mod}_N\}$ of modules where the "entry" module $\texttt{mod}_0$ is assumed to take no arguments. The execution involves solving classical planning subproblems, internal operations on the registers, calls to other modules, and executions of ground actions. The modules do not share memory states nor registers, but they all act on the (external) planning states s.

At each time point during execution, there is a single active module $\texttt{mod}_\ell$ that defines the current set of rules, and there is a current augmented state $(s, m, \boldsymbol{v})$. While no call or do rule is selected, SIW$_\mathrm{M}$ behaves exactly as SIW$^*_\mathrm{R}$. However, if a call rule $(m, C) \mapsto (\texttt{mod}_j(x_1, x_2, \ldots, x_n), m')$ is chosen, where $\texttt{mod}_j$ refers to $\langle \text{args}, Z, M, \mathfrak{R}, \Phi, m_0, R, \rangle$, the following instructions are executed:

1. Push context $(\ell, \boldsymbol{v}, m')$ where $\boldsymbol{v}$ is value for registers,
2. Set value of arguments of $\texttt{mod}_j$ to those given by x_i,
3. Set memory state to m_0 (the initial state of $\texttt{mod}_j$),
4. Set the current set of rules to R,
5. [Cont. execution of $\texttt{mod}_j$ until no rule is applicable], and
6. Pop context $(\ell, \boldsymbol{v}, m')$, set value of registers to $\boldsymbol{v}$, memory state to m', and rules R to those in $\texttt{mod}_\ell$.

If a do rule $(m, C) \mapsto (\texttt{name}(x_1, x_2, \ldots, x_n), m')$ is chosen, on the other hand, then an applicable ground action $\texttt{name}(o_1, o_2, \ldots, o_n)$ at the current state s with the object o_i in x_i, $1 \leq i \leq n$, is applied and the memory state is set to m'. If no such ground action exists, an error code is returned. The implementation of the SIW$_\mathrm{M}$ interpreter and all extended, indexical policies and sketches, and modules that were discussed in the paper are available online (Bonet, Drexler, and Geffner 2024).

Discussion

A basic, concrete question about policy reuse in a planning setting is: can a policy for putting one block on top of another be used for building any given tower of blocks, and eventually any block configuration? The question is relevant because it tells us that policies for building given towers and block configurations do not have to be learned from scratch, but that they can be learned bottom up, from simpler to complex, one after the other (Ellis et al. 2023). The subtlety is that in order for complex policies to use simpler policies, the former must pass the right parameters to the latter depending on the context defined by the top goal and the current state. In this work, we have developed a language for representing such policies and sketches, and this form of hierarchical composition.

We have also addressed another source of complexity when learning general policies and sketches: the complexity of the features involved. We have shown that indexical features whose values depend on the content of registers that can be dynamically loaded with objects, can be used to drastically reduce the complexity of the features needed, in line with the intuition of the so-called deictic representations, where a constant number of "visual marks" are used to mark objects so that they can be easily referred to (Chapman 1989; Ballard et al. 1996; Finney et al. 2013). In our setting,

[3]Block x is *well-placed* in state s iff $on(x, y)$ holds in s when (x, y) is in O, and recursively y is well-placed; it is *misplaced* iff it is not well-placed. In the example, it is also assumed that the lowest block of the target tower must be placed on the table.

Algorithm 3: SIW_M uses set of modules M (extended sketches) for solving a problem P via possibly nested calls, execution of ground actions, and IW searches.

```
 1: Input: Collection $\mathcal{M} = \{\texttt{mod}_0, \texttt{mod}_1, \ldots, \texttt{mod}_N\}$ of modules with entry module $\texttt{mod}_0$
 2: Input: Planning problem $P$ with initial state $s_0$ on which the features in $\Phi$ are well defined
 3: Initialize stack
 4: Let $\mathfrak{R}^j$, $m_0^j$, and $R^j$ be the set of registers, initial memory, and rules of $\texttt{mod}_j$, $j = 0, 1, \ldots, N$
 5: $\ell \leftarrow 0$, $R \leftarrow R^\ell$, and $\bar{s} \leftarrow (s_0, m_0^\ell, \boldsymbol{v})$ for $\boldsymbol{v} \in \text{Obj}(P)^{\mathfrak{R}^\ell}$
 6: while $s$ in $\bar{s} = (s, m, \boldsymbol{v})$ is not a goal state of $P$
 7:   if $m$ is internal memory
 8:     Find rule $r = (m, C) \mapsto (E, m')$ with $s, \boldsymbol{v} \vDash C$
 9:     if $r$ is not found
10:       if stack is empty, return FAILURE                % Stalled
11:       Pop context $(j, \boldsymbol{v}', m')$ from stack
12:       $\ell \leftarrow j$, $R \leftarrow R^\ell$, $m \leftarrow m'$ and $\boldsymbol{v} \leftarrow \boldsymbol{v}'$
13:     else
14:       if Load$(\mathsf{C}, \mathfrak{r})$ in $E$, $\boldsymbol{v}[\mathfrak{r}] \leftarrow o$ for some $o \in \mathsf{C}(s, \boldsymbol{v})$
15:       $\bar{s} \leftarrow (s, m', \boldsymbol{v})$
16:   else                                   % $m$ is external memory
17:     Find call/do rule $r{=}(m, C){\mapsto}(E, m')$ with $s, \boldsymbol{v} \vDash C$
18:     if $r = (m, C) \mapsto (\texttt{mod}_j(x_1, x_2 \ldots, x_n), m')$
19:       Push context $(\ell, \boldsymbol{v}, m')$ into stack
20:       $R \leftarrow R^j$, $m \leftarrow m_0^j$          % Hand control to $\texttt{mod}_j$
21:     elsif $r = (m, C) \mapsto (\texttt{name}(x_1, x_2 \ldots, x_n), m')$
22:       Find ground action $a = \texttt{name}(o_1, o_2, \ldots, o_n)$ applicable at $s$ with $o_i \in x_i(s, \boldsymbol{v})$, $i = 1, 2, \ldots, n$
23:       if there is no such action, return FAILURE
24:       $\bar{s} \leftarrow (s', m', \boldsymbol{v})$ where $(s, a, s')$ is transition in $P$
25:     else                                 % No such rule is found
26:       Run IW search from $s$ to find goal state $s'$ of $P$, or
27:         state $s'$ such that $s' \prec_{r/\boldsymbol{v}} s$ for some (external) rule
28:         $r = (m, C) \mapsto (E, m')$ in $R$
29:       if no such state is found, return FAILURE
30:       $\bar{s} \leftarrow (s', m', \boldsymbol{v})$
31: return path from $s_0$ to the goal state $s$
```

an object can be regarded as marked with "mark" $\mathfrak{r}$ when the object is loaded into register $\mathfrak{r}$.

The use of registers and concept features allows for general policies that map states into ground actions too, even if the set of ground actions change from instance to instance. This is different than general policies defined as filters on state transitions (Bonet and Geffner 2018; Francès, Bonet, and Geffner 2021), which require a model for determining the possible transitions from a state. Policies that map states into actions are more conventional and can be applied model-free.

The achieved expressivity is the result of three extensions in the language of policies and sketches: internal memory states, like in finite state controllers for sequencing behaviors, indexical concepts and features, whose denotation depends on the value of registers that can be updated, and modules that wrap up policies and sketches and allow them to call each other by passing parameters as a function of the state, the registers, and the goals (represented in the state).

The language of extended policies and sketches adds an interface for calling policies and sketches from other policies and sketches, even recursively, as illustrated in the examples. The resulting language has elements in common with programming languages and planning programs (Aguas, Celorrio, and Jonsson 2016; Segovia-Aguas, Jiménez, and Jonsson 2019, 2021), but there are key differences too. In particular, the use of a rich feature language does not limit policies and sketches to deal with classes of problems with a fixed goal and the policy and sketch modules do not have to represent full procedures or policies; they can also represent sketches where "holes" are filled in with polynomial IW searches when the sketches have bounded width.

Provided with this richer language for policies and sketches, the next step is learning them from small problem instances, adapting the methods developed for learning policies and sketches (Francès, Bonet, and Geffner 2021; Drexler, Seipp, and Geffner 2022). The new, extended language also opens the possibility of learning hierarchical policies bottom-up by generating and reusing policies, instead of the more common approach of learning them top-down (Drexler, Seipp, and Geffner 2023). Finally, the study of theoretical properties of the modular framework, like acyclicity and termination, is also important and left for future work.

Acknowledgments

The research of H. Geffner has been supported by the Alexander von Humboldt Foundation with funds from the Federal Ministry for Education and Research. It has also received funding from the European Research Council (ERC), Grant agreement No. No 885107, and Project TAILOR, Grant agreement No. 952215, under EU Horizon 2020 research and innovation programme, the Excellence Strategy of the Federal Government and the NRW Länder, Germany, and the Knut and Alice Wallenberg (KAW) Foundation under the WASP program.

References

Agre, P. E.; and Chapman, D. 1990. What are plans for? *Robotics and Autonomous Systems*, 6: 17–34.

Aguas, J. S.; Celorrio, S. J.; and Jonsson, A. 2016. Generalized planning with procedural domain control knowledge. In *Proc. ICAPS*, 285–293.

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and Patel-Schneider, P. F., eds. 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

Ballard, D. H.; Hayhoe, M. M.; Pook, P. K.; and Rao, R. P. N. 1996. Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences*, 20: 723–742.

Bonet, B.; Drexler, D.; and Geffner, H. 2024. Code and Data for the Paper titled "On Policy Reuse: An Expressive Language for Representing and Executing General Policies that Call Other Policies". https://doi.org/10.5281/zenodo.10814690.

Bonet, B.; and Geffner, H. 2018. Features, Projections, and Representation Change for Generalized Planning. In *Proc. IJCAI*, 4667–4673.

Bonet, B.; and Geffner, H. 2021. General Policies, Representations, and Planning Width. In *Proc. AAAI*, 11764–11773.

Bonet, B.; and Geffner, H. 2023. General Policies, Subgoal Structure, and Planning Width. *arXiv preprint arxiv:2311.05490*.

Bueno, T. P.; de Barros, L. N.; Mauá, D. D.; and Sanner, S. 2019. Deep Reactive Policies for Planning in Stochastic Nonlinear Domains. In *Proc. AAAI*, 7530–7537.

Chapman, D. 1989. Penguins can make cake. *AI magazine*, 10(4): 45–45.

Dietterich, T. G. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13: 227–303.

Drexler, D.; Seipp, J.; and Geffner, H. 2022. Learning Sketches for Decomposing Planning Problems into Subproblems of Bounded Width. In *Proc. ICAPS*, 62–70.

Drexler, D.; Seipp, J.; and Geffner, H. 2023. Learning Hierarchical Policies by Iteratively Reducing the Width of Sketch Rules. In *Proc. KR*, 208–218.

Dumancic, S.; Guns, T.; and Cropper, A. 2021. Knowledge refactoring for inductive program synthesis. In *Proc. AAAI*, 7271–7278.

Ellis, K.; Wong, L.; Nye, M.; Sable-Meyer, M.; Cary, L.; Anaya Pozo, L.; Hewitt, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2023. DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning. *Phil. Trans. R. Soc. A*, 381: 20220050.

Fern, A.; Yoon, S.; and Givan, R. 2006. Approximate policy iteration with a policy language bias: Solving relational Markov decision processes. *Journal of Artificial Intelligence Research*, 25: 75–118.

Finney, S.; Gardiol, N.; Kaelbling, L. P.; and Oates, T. 2013. The Thing That We Tried Didn't Work Very Well : Deictic Representation in Reinforcement Learning. *CoRR*, abs/1301.0567.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning General Planning Policies from Small Examples Without Supervision. In *Proc. AAAI*, 11801–11808.

Garg, S.; Bajpai, A.; and Mausam. 2020. Generalized Neural Policies for Relational MDPs. In *Proc. ICML*.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proc. ICAPS*, 408–416.

Hafner, D.; Lee, K.-H.; Fischer, I.; and Abbeel, P. 2022. Deep hierarchical planning from pixels. In *Proc. NeurIPS*, 26091–26104.

Illanes, L.; and McIlraith, S. A. 2019. Generalized Planning via Abstraction: Arbitrary Numbers of Objects. In *Proc. AAAI*, 7610–7618.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence*, 113: 125–148.

Kulkarni, T. D.; Narasimhan, K.; Saeedi, A.; and Tenenbaum, J. 2016. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Proc. NeurIPS*, 3675–3683.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *Proc. ECAI 2012*, 540–545.

Liu, A.; Xu, H.; Van den Broeck, G.; and Liang, Y. 2023. Out-of-distribution generalization by neural-symbolic joint training. In *Proc. AAAI*, 12252–12259.

Machado, M. C.; Bellemare, M. G.; and Bowling, M. 2017. A Laplacian Framework for Option Discovery in Reinforcement Learning. In *Proc. ICML*, 2295–2304.

Martín, M.; and Geffner, H. 2004. Learning Generalized Policies from Planning Examples Using Concept Languages. *Applied Intelligence*, 20(1): 9–19.

McGovern, A.; and Barto, A. G. 2001. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *Proc. ICML*, 361–368.

Parr, R.; and Russell, S. 1997. Reinforcement Learning with Hierarchies of Machines. In *Proc. NeurIPS*, 1043–1049.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning. In *ICAPS Workshop PRL*, 16–24.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2019. Computing programs for generalized planning using a classical planner. *Artificial Intelligence*, 272: 52–85.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2021. Generalized planning as heuristic search. In *ICAPS*, 569–577.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2): 393–401.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011. Qualitative Numeric Planning. In *Proc. AAAI 2011*, 1010–1016.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proc. ICAPS*, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning Generalized Policies without Supervision Using GNNs. In *Proc. KR*, 474–483.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2023. Learning General Policies with Policy Gradient Methods. In *Proc. KR*, 647–657.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112: 181–211.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research*, 68: 1–68.