General and Reusable Indexical Policies and Sketches

Blai Bonet
Universitat Pompeu Fabra, Spain
bonetblai@gmail.com

Dominik Drexler
Linköping University, Sweden
dominik.drexler@liu.se

Hector Geffner
RWTH Aachen University, Germany
Linköping University, Sweden
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, which is based on collections of rules defined on a set of Boolean and numerical features. In this work, we consider extensions of this basic language aimed at making policies and sketches more flexible and reusable. For this, three basic extensions are considered: 1) internal memory states, as in finite state controllers, 2) indexical features, whose values are a function of the state and a number of internal registers that can be loaded with objects, and 3) modules that wrap up policies and sketches and allow them to call each other by passing parameters. In addition, unlike previously defined policies that select actions indirectly by the selection of state transitions, the new language allows for the selection of actions directly. The expressive power of the resulting language for recombining policies and sketches is illustrated through examples. The problem of learning policies and sketches in the new language is left for future work.

1 Introduction

Classical planners solve problems over large state spaces by exploiting problem structure. Domain-independent methods assume, for example, that subgoals are independent when deriving heuristics [6, 28], or that subproblems have low width [34, 35]. Domain-dependent methods, on the other hand, make problem structure explicit in the form of hierarchies that express how tasks decompose into subtasks [18, 26, 5].

An alternative language for representing problem structure explicitly has been introduced recently in the form of *sketches* [8, 9]. Sketches are collections of rules of the form $C \mapsto E$ 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 [8]. For example, a sketch of width k decomposes problems into subproblems that are solved by the IW algorithm in time exponential in k [34]. 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 [9].

The Delivery domain is a simple variation of the Taxi domain used in RL [15] that is useful for illustrating these notions. In Delivery, packages spread in a grid are to be picked, one by one, and

NeurIPS 2023 Workshop on Generalization in Planning (GenPlan 2023).

delivered to a designated target cell. A sketch of width 2 decomposes the problem of delivering multiple packages into subproblems of delivering just one package. This is expressed with a simple sketch rule $\{n > 0\} \mapsto \{n\downarrow\}$ that involves a single feature n that tracks the number of undelivered packages, and asks for the value of this feature to be decreased. The sketch of width 1 replaces this rule by two rules $\{\neg H, n > 0\} \mapsto \{H\}$ and $\{H\} \mapsto \{\neg H, n\downarrow\}$ where the additional Boolean feature H is true when the agent holds a package. The first rule requires getting hold of a package, if there are still undelivered packages; the second rule requires delivering the package being held. Finally, a sketch of width 0, which expresses a full policy, instructs the agent to decrease the distance p to the nearest package, to pick it up when this distance is 0, to decrease the distance t to the target, and to drop it there when this distance is 0. This policy corresponds to the four rules $\{\neg H, p > 0\} \mapsto \{p\downarrow, t?\}$, $\{\neg H, p = 0\} \mapsto \{H\}$, $\{H, t > 0\} \mapsto \{t\downarrow\}$, and $\{H, t = 0\} \mapsto \{\neg H, n\downarrow, p?\}$. The sketches and policy are fully general for the domain as they can be used for solving instances of any grid size, any number of packages, any initial configuration of the packages, and any target location for them.

A method for learning sketches is developed in [16] where both the rules and its features are obtained by solving a combinatorial optimization problem that accepts three inputs: an upper bound on the width of the sketch to be learned, a small number of domain instances, and a pool of features defined from the domain predicates in a domain-independent manner. The learning method does not guarantee that the learned sketches are general and correct, and hence split the infinite collection $\mathcal{Q}$ of target problems P into subproblems of width no greater than the bound, yet the sketches learned for a number of benchmark domains have been shown to be correct in this sense.

In this work, we consider extensions of the basic language of sketches and policies (i.e., sketches of width 0) that render them more flexible and reusable. For this, three basic extensions are introduced: *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 package policies and sketches in a way that they can be reused by other policies or sketches.[1] For example, an existing policy π_1 for getting hold of the object in register $\mathtt{r}$ can be reused by a policy π for delivering some package to the target cell.

The current language of policies and sketches does not support the *reuse* of existing policies and sketches, and assumes instead that the top goals are set up externally.[2] For *reusing* policies, the policy and sketch language must allow goals to be set up internally as well; e.g., for reusing a policy π_{on} that manages to achieve an atom $on(x, y)$ for any pair of blocks x and y in a policy π_t for building any tower, it is necessary for the policy π_t to invoke policy π_{on} on suitable pairs of arguments x and y. The registers, and the indexical features and concepts that use them, provide the right interface for recombining policies and sketches in this way.

The paper is structured as follows. Sections 2 and 3 cover related work and review planning and sketches. Section 4 introduces two of the proposed extensions, Section 5 presents their formal syntax and semantics, and Section 6 defines modules, call rules, and action calls.

2 Related Work

General policies. The paper builds on a research thread that introduced the notions of width, generalized rule-based policies and sketches of bounded width, as well as methods for learning them [34, 7–9, 24, 16, 17]. The problem of representing and learning general policies has a long history [32, 38, 21], and general plans have also been formulated in terms of first-order logic [44, 30], derived by forms of first-order regression [11, 52, 51, 42], and represented and learned using neural nets [27, 50, 12, 41, 25, 48]. A limitation of these approaches is the lack of a language to accommodate policies that may call other policies by passing parameters; a limitation that is addressed in this work.

Deictic representations. The use of registers to store objects in the formal language of sketches is closely related to the use of indices and visual markers in deictic or indexical representations [13, 1, 4]. The computational value of such representations, however, has not been clear, and more recent work has shown that certain deictic representations in RL actually harm performance [22].

[1] An implementation of all three extensions into the Mimir planning system [47] is available online [10].

[2] 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 $p_G(c)$ true in s means that that $p(c)$ must be true in the goal.

Algorithm 1: SIW_R search given sketch R

Input: Sketch R (set of rules) over features in Φ that defines the relation $\prec_R$
Input: Planning problem P with initial state s_0 in which the features in Φ are well defined
Set state $s \leftarrow s_0$
While the state s is not a goal in P:
    Do IW search from s to find s' that is either a goal state in P, or $s' \prec_R s$
    If s' is not found, return FAILURE Subproblem $P[s]$ is unsolvable
    Set $s \leftarrow s'$
Return path from s_0 to the goal state s

Figure 1: SIW_R: sketch R used to decompose problem into subproblems, each solved with IW.

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 in a suitable way.

Hierarchical and goal-conditioned RL. Hierarchical structures have been used in RL in the form of options [49], hierarchies of machines [40] and MaxQ hierarchies [15], among others. While this "control knowledge" is often provided by hand, a vast literature has explored methods for learning them based on "bottleneck states" [39], "eigenpurposes" of the matrix dynamics [37], and informal width-based considerations [31, 2]. The use of states extended with a goal encoding is common in goal-conditioned RL [36] and some hierarchical RL approaches [33], yet the invocation of subpolicies is usually assumed to involve no parameters at all.

Intrinsic rewards and reward machines. Subgoal structure in RL has also been represented by means of intrinsic rewards [43, 54] and reward machines [29, 14], which are closely related to sketches. Three differences are that sketches are defined in terms of state features, not additional variables, so there is no need for cross-products; sketches have a theory of width that tells us where problems have to be split into subproblems; and finally, sketches have a notion of termination that ensures that subgoaling does not result in cycles [9].

3 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 the same 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, with non-negative integer values [8, 9]. 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 value for the features 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.

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 Fig 1 uses this problem decomposition for solving problems P in $\mathcal{Q}$ by solving subproblems $P[s]$ via the IW algorithm [34]. 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 [8, 9, 46].

3.1 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 $\text{IW}(k)$, which is a simple breadth-first search where newly generated states are pruned if they do not make a tuple (set) of k atoms or less true for the first time in the search [34]. If the width of P is bounded but the value of the bound k is not known, a plan (not necessarily optimal) can be found by running the

algorithm IW with the same complexity bounds. IW calls IW(i) iteratively with $i = 0, \ldots, k$, until P is solved [34]. 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 resulting subproblems $P[s]$ have all 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 $s_n = s_0$. R is acyclic in $\mathcal{Q}$ if it is acyclic in each problem P in $\mathcal{Q}$. The SIEVE algorithm [46] can check whether a sketch R is **terminating**, and hence acyclic, by just considering the rules in R and the graph that they define [9]. 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** [8].

Example 1 defines an acyclic sketch of width 2 for achieving the atom $on(x, y)$ for two blocks x and y in any Blocks world instance.

Example 1: Sketch for class $\mathcal{Q}_{on}$ of width 2

A sketch R for the class $\mathcal{Q}_{on}$ of all Blocksworld problems with atomic goal $on(x, y)$ for two blocks x and y can be defined by means of a numerical feature D that counts the number of blocks above x or y, and a feature *On* that represents whether x is on y. The set of features is thus $\Phi = \{On, \mathsf{D}\}$, and the rules in R are:

$\{\mathsf{D} > 0\} \mapsto \{\mathsf{D}{\downarrow}\}$	Put away a block from above x or y
$\{\mathsf{D} = 0, \neg On\} \mapsto \{On, \mathsf{D}?\}$	Put the block x on top of block y

The first rule says that in states where D is positive, the states where D is lower are possible subgoals; the second rule, that in states where D is zero, the subgoal is to make the feature *On* true. The first subproblems have width 2, while the second subproblem has width 1.

4 Extended Sketches

The first two extensions of sketches are introduced.

4.1 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 the current state and memory are s and m 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 certain cases, a rule like $(m, C) \mapsto (true, m')$, where *true* is a feature that is true in all states, allows a transition in the memory state from m to m' without involving also a transition from a state s into a different state s'. These rules are abbreviated as $(m, C) \mapsto (\{\}, m')$.

4.2 Parametric Features, Concepts, Roles, and Registers

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 means of two new classes of features called *concepts* and *roles* that denote sets of objects and set of object pairs, respectively. There is no need to commit to a specific language to describe concepts and roles, yet such languages, often based on description logics [3], are common in planning [38, 20, 53, 7, 23, 24, 19, 16]. 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' in s. 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 numerical feature is given by the cardinality of the concept C or role R in the state; namely, the number of objects and 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 change may affect the denotation of indexical features that depend on the value of the register. 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 or 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 which is *non-deterministic*. 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.

Rules with a load effect are called *internal rules*, as they capture changes in the internal memory only, while the other rules are called *external rules*. For simplicity, it is assumed, that each load rule contains a single load effect, and that load rules have their own memory state, meaning that no external rule can be applied in the same memory where a load rule is applied. Rules $(m, C) \mapsto (true, m')$, abbreviated $(m, C) \mapsto (\{\}, m')$, are also internal rules. Memory states associated with internal rules are referred to as *internal memory*; others, as *external memory*.

Example 2 defines a general indexical policy for clearing multiple blocks. The set of blocks to be cleared is expressed with the concept C, while $\mathsf{N} \subseteq \mathsf{C}$ is the set of blocks in C that are not clear in the current state. The policy can be understood as putting a *"mark"* $\mathfrak{r}_0$ in a block in N, and a second *mark* $\mathfrak{r}_1$ in the topmost block above the one in $\mathfrak{r}_0$. The block marked in $\mathfrak{r}_1$ is then put away and the "mark" $\mathfrak{r}_1$ is updated. When the block marked in $\mathfrak{r}_0$ becomes clear, the process repeats until the counter N is decreased to zero. It is important to notice that the standard language for representing policies can express a policy for this task using a non-indexical numerical feature that counts the number of blocks above *any* block in C, but *it cannot express this particular policy*.

Example 2: General indexical policy for the class $\mathcal{Q}_{clear^*}$

The class $\mathcal{Q}_{clear^*}$ contains Blocksworld problems whose goal is a conjunction of $clear(x)$ atoms. For an instance of this class, let C be the concept that contains all blocks that need to be cleared. A general policy π_{clear^*} for $\mathcal{Q}_{clear^*}$ can be obtained with five memory states, the registers $\mathfrak{r}_0$ and $\mathfrak{r}_1$, the concept N for the blocks in C that are not clear, the indexical concept T for the topmost block above the block in $\mathfrak{r}_0$, the Boolean H that is true iff a block is being held, and the Boolean A that is true if the block in $\mathfrak{r}_1$ is above some block in C. The set of features is $\Phi = \{H, A, \mathsf{N}, \mathsf{T}\}$, the initial memory state is m_0, and the rules are:

% External rules

$m_0 \parallel \{H\} \mapsto \{\neg H, \mathsf{N}?\} \parallel m_1$	If holding, put block away, move to m_1
$m_0 \parallel \{\neg H\} \mapsto \{\} \parallel m_1$	Else, skip to m_1

% Internal rules

$m_1 \parallel \{\mathsf{N} > 0\} \mapsto \{\mathit{Load}(\mathsf{N}, \mathfrak{r}_0), \mathsf{T}?\} \parallel m_2$	Put a block to be cleared into register $\mathfrak{r}_0$
$m_2 \parallel \{\mathsf{T} > 0\} \mapsto \{\mathit{Load}(\mathsf{T}, \mathfrak{r}_1), A?\} \parallel m_3$	Put topmost block above $\mathfrak{r}_0$ into register $\mathfrak{r}_1$
$m_2 \parallel \{\mathsf{T} = 0\} \mapsto \{\} \parallel m_1$	If block in $\mathfrak{r}_0$ clear, go to m_1 to select another one

% External rules

$m_3 \parallel \{\neg H, A\} \mapsto \{H, \neg A, \mathsf{N}?, \mathsf{T}?\} \parallel m_4$	Pick (the block in) $\mathfrak{r}_1$
$m_4 \parallel \{H\} \mapsto \{\neg H\} \parallel m_2$	Put $\mathfrak{r}_1$ away from the blocks in C
$m_4 \parallel \{H\} \mapsto \{\neg H, \mathsf{N}{\downarrow}\} \parallel m_2$	Put $\mathfrak{r}_1$ (which is in N) away from the blocks in C

5 Syntax and Semantics of Extended Sketches

A feature ϕ for a problem P is a state function that assigns a value to each reachable state in P. The feature is **Boolean** if it returns a truth value, **numerical** if it returns a non-negative integer, **concept** if it returns a set of objects, and a **role** if it returns a set of object pairs. A concept feature C (resp. role feature R) may be used in rules as a numerical feature, in which case, $\mathsf{C}=0$ or $\mathsf{C}>0$ (resp. $\mathsf{R}=0$ or $\mathsf{R}>0$) refers to whether its denotation is empty, and $\mathsf{C}{\downarrow}$ and $\mathsf{C}{\uparrow}$ (resp. $\mathsf{R}{\downarrow}$ and $\mathsf{R}{\uparrow}$) refer to changes in the size of its denotation.

A feature is parametric if it refers to a register; i.e., if its value in a state depends on the value of a register. A parametric feature may be Boolean, numerical, a concept, or a role.

Definition 2 (Extended 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 set of conditions of the form p, $\neg p$, $n=0$, and $n>0$ for Boolean and numerical features p and n in Φ, and E is a set of effects of the form p, $\neg p$, $p?$ for Boolean p, $n{\downarrow}$, $n{\uparrow}$, and $n?$ for numerical n, or it has a single load effect of form Load$(\mathsf{C}, \mathfrak{r})$ for some concept C and register $\mathfrak{r}$. If E contains a Load$(\mathsf{C}, \mathfrak{r})$, it also contains $\phi?$ for the features ϕ in $\Phi(\mathfrak{r})$, but no other effect. In such a case, the memory m must be internal, and the rule is called a load or internal rule. If m is external memory, r is called an external rule.*

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. An extended sketch is well defined on a class $\mathcal{Q}$ if its set of features Φ is well defined on $\mathcal{Q}$. If m is a memory state, $R(m)$ denote the subset of rules in R of form $(m, C) \mapsto (E, m')$.*

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 in M, and $\boldsymbol{v}$ is a vector of objects in Obj$(P)^{\mathfrak{R}}$ that tells the content of each register $\mathfrak{r}$, denoted as $\boldsymbol{v}[\mathfrak{r}]$.*

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

*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 $\boldsymbol{v}$.*

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

Load (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 in M. The pair $\langle (s, m, \boldsymbol{v}), (s, m', \boldsymbol{v}')\rangle$ is a **reduction step** if there is an internal rule r in R of form $(m, C) \mapsto (E, m')$ such that 1) s satisfies the condition C given $\boldsymbol{v}$, and 2) $\boldsymbol{v}'$ is equal to $\boldsymbol{v}$, except if Load$(\mathsf{C}, \mathfrak{r})$ is in E, in which case $\boldsymbol{v}'[\mathfrak{r}] \in \mathsf{C}(s, \boldsymbol{v})$. 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 $(s, m, \boldsymbol{v}) \rightarrow^* (s, m', \boldsymbol{v}')$. For convenience, if m is external, we also write $(s, m, \boldsymbol{v}) \rightarrow^* (s, m, \boldsymbol{v})$.*

An **initial augmented state** for P is of the form $(s_0, m, \boldsymbol{v})$ where s_0 is the initial state in P, and $(s_0, m_0, \boldsymbol{v}_0) \rightarrow^* (s_0, m, \boldsymbol{v})$ 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 either their memory or the contents of the registers. Each such initial augmented state defines an **initial subproblem** $P[s_0, m, \boldsymbol{v}]$ per 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 such that $s' \prec_{r/\boldsymbol{v}} s$ 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.*
2. *Subproblem $P[s', m'', \boldsymbol{v}']$ is **induced** by subproblem $P[\bar{s}]$ if $r = (m, C) \mapsto (E, m')$, m' is internal, and $(s', m', \boldsymbol{v}) \rightarrow^* (s', m'', \boldsymbol{v}')$.*

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.

By "jumping" over subproblems $P[s, m, v]$ where m is internal, the definition ensures that the subproblems that make it into P° all have memory states m that are external, and hence represent classical planning problems. 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 $(s, m, \boldsymbol{v}) \rightarrow^* (s, m', \boldsymbol{v}')$. 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 a non-negative integer k, denoted by $w_R(P) \leq k$, if the width of each subproblem in P° is bounded by k. Likewise, 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 problem P in $\mathcal{Q}$.*

This **serialized width** is zero for the indexical sketch in Example 2, as the sketch represents a policy where each subproblem is solved in a single step.

5.1 Termination for Extended Sketches

Termination is a key property for sketches R as it guarantees that R is acyclic on *any* problem P where it defines a polynomial number of subproblems [9]. Termination can be tested in polynomial time by a suitable adaptation of the SIEVE algorithm [45, 9]. If the extended sketch is reducible, terminating, and has bounded serialized width over a class $\mathcal{Q}$, then any problem P in $\mathcal{Q}$ can be solved in polynomial time with the algorithm SIW^*_{R} shown in Fig. 2.

For an extended sketch $\langle M, \mathfrak{R}, \Phi, m_0, R\rangle$, the graph $G^*(R)$ that is processed by SIEVE is a labeled and directed graph where the vertices are each of the $|M| \times 2^{|\Phi|}$ pairs (m, ν), where m is a memory state and ν is a valuation for the conditions p and $n = 0$ for the Boolean and numerical/concept/role features p and n in Φ, respectively. The edge set contains edges $e = \langle (m, \nu), (m', \nu')\rangle$ if there is a rule $r = (m, C) \mapsto (E, m')$ in R such that ν is consistent with C and (ν, ν') is compatible with the effect E. The edge label $\ell(e)$ is given by the union of the effects E in such rules Once the graph $G^*(R)$ is constructed, an slight variant of the SIEVE algorithm is run to either accept or reject $G^*(R)$ [9]. In the former case, we say that the extended sketch is **terminating.** Notice that registers and load effects play no role in determining termination, as indeed, the effect of register updates on indexical features X is expressed in load rules by means of expressions like X?. A terminating (extended) sketch R is **acyclic** on any problem, meaning that there cannot be cyclic sequences of augmented states complying with the rules of R. In addition, terminating sketches result in a polynomial number of subproblems [9] from which the following result can be established:

Theorem 10. *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^*_R finds plans for any problem P in $\mathcal{Q}$ in polynomial time.*

6 Wrapping Policies and Sketches into Modules for Reuse

For reusing policies and sketches, they are wrapped into *modules*. 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

Algorithm 2: SIW^*_R search with extended sketch R

Input: Extended sketch $\langle M, \mathfrak{R}, \Phi, m_0, R\rangle$ that defines the relation $\prec_{R/\boldsymbol{v}}$
Input: Planning problem P with initial state s_0 in $\mathcal{Q}$ on which the features in Φ are well defined
Set augmented state $\bar{s} \leftarrow (s, m, \boldsymbol{v})$ for $s = s_0$, $m = m_0$, and some $\boldsymbol{v} \in \text{Obj}(P)^{\mathfrak{R}}$
While the state s in $\bar{s}$ is not a goal in P:
    If m is internal memory:
        Find internal rule $r = (m, C) \mapsto (E, m')$ such that C is satisfied by s
        If the is no such rule, return FAILURE (The sketch is not reducible on P)
        Set $\boldsymbol{v}[\mathfrak{r}] \leftarrow o$ for some object o in $\mathsf{C}(s, \boldsymbol{v})$, if $\mathit{Load}(\mathsf{C}, \mathfrak{r})$ is the load effect in E
        Set $\bar{s} = (s, m', \boldsymbol{v})$
    Else:
        Do IW search from s to find s' that is either a goal state in P,
          or $s' \prec_{r/\boldsymbol{v}} s$ for some (external) rule $r = (m, C) \mapsto (E, m')$
        If s' is not found, return FAILURE (Subproblem $P[s, m, \boldsymbol{v}]$ is unsolvable)
        Set $\bar{s} = (s', m', \boldsymbol{v})$
Return path from s_0 to the goal state s

Figure 2: SIW^*_R solves a problem P by using extended sketch R to decompose P into subproblems solved with an IW search. Completeness of SIW^*_R captured in Theorem 10.

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 definition may depend on the value of the arguments and the features in Z. The sketch R in a module can contain two new types of internal rules, *call and do rules,* that permit to call other modules, and to directly execute ground actions of the planning problem, respectively. The value for the arguments in either case are 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 or to provide values to arguments in call and do rules. If the name of the module is `name`, we refer to it as $\texttt{name}(x_1, x_2, \ldots, x_n)$.

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 3 shows the module $\texttt{mclear}(\mathsf{C})$ for the policy in Example 2, and a module $\texttt{on}(\mathsf{X}, \mathsf{Y})$ for the Example 1 that reuses `mclear`. Example 4 shows the module $\texttt{tower}(\mathsf{O}, \mathsf{X})$ for building a given tower of blocks, expressed by the object pairs in O to put in place on top of the block X. Finally, Example 5 shows a sketch module $\texttt{blocks}(\mathsf{O})$ for solving arbitrary instances of Blocksworld.

Example 3: Modules $\texttt{mclear}(\mathsf{C})$ and $\texttt{on}(\mathsf{X}, \mathsf{Y})$

The module $\texttt{mclear}(\mathsf{C})$ is the tuple $\langle\langle\mathsf{C}\rangle, Z, M, \mathfrak{R}, \Phi, m_0, R\rangle$ where $Z = \emptyset$, $M = \{m_0, m_1, \ldots, m_4\}$, $\mathfrak{R} = \{\mathfrak{r}_0, \mathfrak{r}_1\}$, $\Phi = \{H, A, \mathsf{N}, \mathsf{T}\}$, and R is the set of rules in Example 2.

The module $\texttt{on}(\mathsf{X}, \mathsf{Y})$ is the tuple $\langle\langle\mathsf{X}, \mathsf{Y}\rangle, Z, M, \mathfrak{R}, \Phi, m_0, R\rangle$ where X and Y are singleton concepts denoting the block x and y, $Z = \emptyset$, $M = \{m_0, m_1\}$, $\mathfrak{R} = \emptyset$, $\Phi = \{\mathit{On}\}$, and R is the set of rules:

% Module $\texttt{on}(\mathsf{X}, \mathsf{Y})$

$m_0 \parallel \{\neg \mathit{On}\} \mapsto \texttt{mclear}(\mathsf{X} \cup \mathsf{Y}) \parallel m_1$	Clear the blocks in $\mathsf{X} \cup \mathsf{Y}$, if not already
$m_1 \parallel \{\neg \mathit{On}\} \mapsto \{\mathit{On}\} \parallel m_1$	Put the block x on top of block y

Example 4: Module $\texttt{tower}(\mathsf{O},\mathsf{X})$ for building a single tower of blocks (class $\mathcal{Q}_{tower}$)

The module $\texttt{tower}(\mathsf{O},\mathsf{X})$ is aimed at the class $\mathcal{Q}_{tower}$ of problems where blocks are to be stacked in a *single tower* on the table by achieving the goals $\wedge_{i=1}^{k} on(x_i, x_{i-1})$ 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 $\{(x_i, x_{i-1}) \mid i = 1, ..., k\}$, and X is a concept argument that denotes the lowest block in the target tower that is misplaced.[a] 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 in R are:

% Module $\texttt{tower}(\mathsf{O},\mathsf{X})$

$m_0 \parallel \{\mathsf{X} > 0\} \mapsto \{\mathit{Load}(\mathsf{X}, \mathfrak{r}_0), \mathsf{M}?, \mathsf{W}?\} \parallel m_1$	Put lowest misplaced block in $\mathfrak{r}_0$
$m_1 \parallel \{\mathsf{W} = 0\} \mapsto \texttt{on-table}(\mathfrak{r}_0) \parallel m_2$	Call module $\texttt{on-table}(\mathfrak{r}_0)$ to place $\mathfrak{r}_0$ on the table
$m_1 \parallel \{\mathsf{W} > 0\} \mapsto \texttt{on}(\mathfrak{r}_0, \mathsf{W}) \parallel m_2$	Call module $\texttt{on}$ to place $\mathfrak{r}_0$ on W
$m_2 \parallel \{\mathsf{M} > 0\} \mapsto \texttt{tower}(\mathsf{O}, \mathsf{M}) \parallel m_3$	Recursive call from block that is to be on $\mathfrak{r}_0$

[a] A block x is *well-placed* in a state s iff x is on y if the pair (x, y) is in O, and recursively, y is well-placed. A block 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.

Example 5: Module $\texttt{blocks}(\mathsf{O})$ for arbitrary towers (class $\mathcal{Q}_{blocks}$)

The module $\texttt{blocks}(\mathsf{O})$ is aimed at the class $\mathcal{Q}_{blocks}$ of problems for building many target towers. The module takes a single role argument O whose denotation encodes the pairs (x, y) corresponding to the target $on(x, y)$ atoms as in Example 4. 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 is the concept that contains the lowest misplaced blocks in O.

% Module $\texttt{blocks}(\mathsf{O})$

$m_0 \parallel \{\mathsf{L} > 0\} \mapsto \{\mathit{Load}(\mathsf{L}, \mathfrak{r}_0)\} \parallel m_1$	Load a lowest misplaced block into $\mathfrak{r}_0$
$m_1 \parallel \{\} \mapsto \texttt{tower}(\mathsf{O}, \mathfrak{r}_0) \parallel m_0$	Build tower starting with block in $\mathfrak{r}_0$

The module $\texttt{blocks}(\mathsf{O})$ calls the **recursive** $\texttt{tower}(\mathsf{O},\mathsf{X})$ which also calls $\texttt{on-table(X)}$ for putting a block on the table (not spelled out), and $\texttt{on}(\mathsf{X},\mathsf{Y})$, which calls $\texttt{mclear}(\mathsf{C})$. The result is not a **hierarchical policy** but a **hierarchical sketch** as the use of the module $\texttt{blocks}(\mathsf{O})$ and its submodules involves an IW search in the context of a suitable SIW_R algorithm (below). The IW search, however, does not make calls beyond IW(1) as the width of all these modules is bounded by 1; which is the width of the sketch module $\texttt{on}$ ($\texttt{mclear}$ is a policy module).

6.1 Execution Model: SIW_M

The execution model for modules is captured by the SIW_M algorithm, shown in Fig. 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 may involve solving classical planning subproblems, internal operations on the registers, calls to other modules, or execution of ground actions. The modules do not share memory states nor registers, but they may all act on the planning states s.

At each time point during the 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_M behaves exactly as SIW^*_R. However, if a call rule $(m, C) \mapsto (\texttt{mod}_i(x_1, x_2, \ldots, x_n), m')$ is chosen, where $\texttt{mod}_i$ refers to $\langle \text{args}, Z, M, \mathfrak{R}, \Phi, m_0, R, \rangle$, the following steps are done:

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

Algorithm 3. SIW_M: Execution Model For Extended Sketches as Modules

Input: Collection $\mathcal{M} = \{\texttt{mod}_0, \texttt{mod}_1, \ldots, \texttt{mod}_N\}$ of modules with entry module $\texttt{mod}_0$
Input: Planning problem P with initial state s_0 in $\mathcal{Q}$ on which the features in Φ are well defined
Initialize stack
Let $\mathfrak{R}^0$, m_0^0, and R^0 be the set of registers, entry point, and rules of $\texttt{mod}_0$
Set $\ell \leftarrow 0$, $R \leftarrow R^\ell$, and $\bar{s} \leftarrow (s, m, \boldsymbol{v})$ for $s = s_0$, $m = m_0^\ell$, and $\boldsymbol{v} \in \text{Obj}(P)^{\mathfrak{R}^\ell}$
While the state s in $\bar{s} = (s, m, \boldsymbol{v})$ is not a goal in P:
    If there is no rule $r = (m, C) \mapsto (E, m')$ such that C is satisfied at s:
        If stack is empty, **raise** FAILURE (Stalled execution at non-goal state)
        Pop context $(j, \boldsymbol{v}', m')$ from stack
        Set $\ell \leftarrow j$, $R \leftarrow R^\ell$, $m \leftarrow m'$ and $\boldsymbol{v} \leftarrow \boldsymbol{v}'$ (Control is back at module $\texttt{mod}_j$)
    Else:
        Find rule $r = (m, C) \mapsto (E, m')$ such that C is satisfied at s
        If $r = (m, C) \mapsto (\texttt{mod}_j(x_1, x_2, \ldots, x_n), m')$ is a call rule:
            Push context $(\ell, \boldsymbol{v}, m')$ into stack
            Set $R \leftarrow R^j$ and $m \leftarrow m_0^j$ (Hand over control to module $\texttt{mod}_j$)
        Else if $r = (m, C) \mapsto (\texttt{name}(x_1, x_2, \ldots, x_n), m')$ is a do rule:
            Find ground action $a = \texttt{name}(o_1, o_2, \ldots, o_n)$ applicable at s such that $o_i \in x_i$
            If there is no such action, **raise** FAILURE (Do rule cannot be fullfilled)
            Set $\bar{s} = (s', m', \boldsymbol{v})$ where (s, a, s') is a transition in P
        Else if r is a load rule with effect $\mathit{Load}(\mathsf{C}, \mathfrak{r})$ in E:
            Set $\boldsymbol{v}[\mathfrak{r}] \leftarrow o$ for some object o in $\mathsf{C}(s, \boldsymbol{v})$
            Set $\bar{s} = (s, m', \boldsymbol{v})$
        Else:
            Do IW search from s to find s' that is either a goal state in P,
              or $s' \prec_{r/\boldsymbol{v}} s$ for some (external) rule $r = (m, C) \mapsto (E, m')$ in R
            If s' is not found, **raise** FAILURE (The width of $P[s, m, \boldsymbol{v}]$ is ∞)
            Set $\bar{s} = (s', m', \boldsymbol{v})$
Return path from s_0 to the goal state s

Figure 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.

Similarly, a do rule $(m, C) \mapsto (\texttt{name}(x_1, x_2, \ldots, x_n), m')$ is chosen, 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'. We have implemented the procedure SIW_M and have run it over all the modules described in the paper. The code and the examples are available online [10]

7 Discussion

We have introduced three language extensions to make policies and sketches more expressive and reusable: internal memory states, like in finite state controllers, 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. 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 some examples, leading to hierarchical policies and sketches. The resulting language has elements in common with programming languages like memory states and registers, but there are key differences too. In particular, the augmented states do not contain just memory states and register values but an external state that is affected by the domain (external) actions. At the same time, the resulting modules do not have to represent full procedures or policies; they can also represent sketches where "holes" are filled by a polynomial IW search 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 standard sketches [16]. In addition, we want to learn hierarchical policies bottom-up by generating and reusing policies, instead of learning them top-down as in [17].

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. The research 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, and the Knut and Alice Wallenberg (KAW) Foundation under the WASP program. Resources were also provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at the National Supercomputer Centre at Linköping University partially funded by the Swedish Research Council through grant agreements no. 2022-06725 and no. 2018-05973.

References

[1] Philip E. Agre and David Chapman. What are plans for? *Robotics and Autonomous Systems*, 6: 17–34, 1990.

[2] Benjamin Ayton and Masataro Asai. Is policy learning overrated?: Width-based planning and active learning for atari. pages 547–555, 2022.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[4] Dana H. Ballard, Mary M. Hayhoe, Polly K. Pook, and Rajesh P. N. Rao. Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences*, 20:723–742, 1996.

[5] Pascal Bercher, Ron Alford, and Daniel Höller. A survey on hierarchical planning – one abstract idea, many concrete realizations. In *Proc. IJCAI 2019*, pages 6267–6275, 2019.

[6] Blai Bonet and Héctor Geffner. Planning as heuristic search. *AIJ*, 129(1):5–33, 2001.

[7] Blai Bonet and Hector Geffner. Features, projections, and representation change for generalized planning. In *Proc. IJCAI 2018*, pages 4667–4673, 2018.

[8] Blai Bonet and Hector Geffner. General policies, representations, and planning width. In *Proc. AAAI 2021*, pages 11764–11773, 2021.

[9] Blai Bonet and Hector Geffner. General policies, subgoal structure, and planning width. *arXiv preprint arxiv:2311.05490*, 2023.

[10] Blai Bonet, Dominik Drexler, and Hector Geffner. Code for the paper titled "General and Reusable Indexical Policies and Sketches". `https://doi.org/10.5281/zenodo.10252342`, 2023.

[11] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *Proc. IJCAI 2001*, pages 690–700, 2001.

[12] Thiago P. Bueno, Leliane N. de Barros, Denis D. Mauá, and Scott Sanner. Deep reactive policies for planning in stochastic nonlinear domains. In *Proc. AAAI 2019*, pages 7530–7537, 2019.

[13] David Chapman. Penguins can make cake. *AI magazine*, 10(4):45–45, 1989.

[14] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. Foundations for restraining bolts: Reinforcement learning with ltlf/ldlf restraining specifications. In *ICAPS*, volume 29, pages 128–136, 2019.

[15] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[16] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proc. ICAPS 2022*, pages 62–70, 2022.

[17] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning hierarchical policies by iteratively reducing the width of sketch rules. In *Proc. KR 2023*, 2023.

[18] Kutluhan Erol, James A. Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proc. AAAI 1994*, pages 1123–1128, 1994.

[19] Patrick Ferber, Liat Cohen, Jendrik Seipp, and Thomas Keller. Learning and exploiting progress states in greedy best-first search. In *Proc. IJCAI 2022*, pages 4740–4746, 2022.

[20] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *Proc. ICAPS 2004*, pages 191–198, 2004.

[21] Alan Fern, Sungwook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence Research*, 25:75–118, 2006.

[22] Sarah Finney, Natalia Gardiol, Leslie Pack Kaelbling, and Tim Oates. The thing that we tried didn't work very well : Deictic representation in reinforcement learning. *CoRR*, abs/1301.0567, 2013. URL `http://arxiv.org/abs/1301.0567`.

[23] Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning. In *Proc. IJCAI 2019*, pages 5554–5561, 2019.

[24] Guillem Francès, Blai Bonet, and Hector Geffner. Learning general planning policies from small examples without supervision. In *Proc. AAAI 2021*, pages 11801–11808, 2021.

[25] Sankalp Garg, Aniket Bajpai, and Mausam. Generalized neural policies for relational mdps. In *Proc. ICML*, 2020.

[26] Ilche Georgievski and Marco Aiello. HTN planning. *AIJ*, 222:124–156, 2015.

[27] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. In *Proc. ICAPS*, 2018.

[28] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[29] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *JAIR*, 73:173–208, 2022.

[30] León Illanes and Sheila A. McIlraith. Generalized planning via abstraction: Arbitrary numbers of objects. In *Proc. AAAI*, pages 7610–7618, 2019.

[31] Miquel Junyent, Vicenç Gómez, and Anders Jonsson. Hierarchical width-based planning and learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 519–527, 2021.

[32] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113: 125–148, 1999.

[33] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29:3675–3683, 2016.

[34] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *Proc. ECAI 2012*, pages 540–545, 2012.

[35] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proc. AAAI 2017*, pages 3590–3596, 2017.

[36] Minghuan Liu, Menghui Zhu, and Weinan Zhang. Goal-conditioned reinforcement learning: Problems and solutions. In *Proc. IJCAI 2022*, pages 5502–5511, 2022.

[37] Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. A Laplacian framework for option discovery in reinforcement learning. In *Proc. ICML 2017*, pages 2295–2304, 2017.

[38] Mario Martín and Hector Geffner. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20(1):9–19, 2004.

[39] Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proc. ICML 2001*, pages 361–368, 2001.

[40] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Proc. NeurIPS*, pages 1043–1049, 1997.

[41] Or Rivlin, Tamir Hazan, and Erez Karpas. Generalized planning with deep reinforcement learning. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*, pages 16–24, 2020.

[42] Scott Sanner and Craig Boutilier. Practical solution techniques for first-order MDPs. *Artificial Intelligence*, 173(5–6):748–788, 2009.

[43] Satinder P. Singh, Richard L. Lewis, Andrew G. Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2:70–82, 2010.

[44] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A new representation and associated algorithms for generalized planning. *AIJ*, 175(2):393–401, 2011.

[45] Siddharth Srivastava, Shlomo Zilberstein, Neil Immerman, and Hector Geffner. Qualitative numeric planning. In *AAAI*, 2011.

[46] Siddharth Srivastava, Shlomo Zilberstein, Neil Immerman, and Hector Geffner. Qualitative numeric planning. In *Proc. AAAI 2011*, pages 1010–1016, 2011.

[47] Simon Ståhlberg. Lifted successor generation by maximum clique enumeration. In *Proc. ECAI 2023*, pages 2194–2201, 2023.

[48] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning general policies with policy gradient methods. In *Proc. KR 2023*, 2023.

[49] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *AIJ*, 112:181–211, 1999.

[50] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *AAAI*, 2018.

[51] Martijn van Otterlo. Solving relational and first-order logical markov decision processes: A survey. In *Reinforcement Learning*, pages 253–292. Springer, 2012.

[52] Chenggang Wang, Saket Joshi, and Roni Khardon. First order decision diagrams for relational MDPs. *Journal of Artificial Intelligence Research*, 31:431–472, 2008.

[53] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *JMLR*, 9:683–718, 2008.

[54] Zeyu Zheng, Junhyuk Oh, Matteo Hessel, Zhongwen Xu, Manuel Kroiss, Hado van Hasselt, David Silver, and Satinder Singh. What can learned intrinsic rewards capture? In *Proc. ICML 2020*, pages 11436–11446, 2020.