PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning

by

Tan Zhi-Xuan

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author .....

Department of Electrical Engineering and Computer Science

January 26, 2022

Certified by .....

Vikash K. Mansinghka

Principal Research Scientist, Brain and Cognitive Sciences

Thesis Supervisor

Certified by .....

Joshua B. Tenenbaum

Professor of Brain and Cognitive Sciences

Thesis Supervisor

Accepted by .....

Leslie A. Kolodziejski

Professor of Electrical Engineering and Computer Science

Chair, Department Committee on Graduate Students

PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning

by

Tan Zhi-Xuan

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The Planning Domain Definition Language (PDDL) is a formal specification language for symbolic planning problems and domains that is widely used by the AI planning community. However, most implementations of PDDL are closely tied to particular planning systems and algorithms, and are not designed for interoperability or modular use within larger AI systems. This limitation makes it difficult to support extensions to PDDL without implementing a dedicated planner for that extension, inhibiting the generality, reach, and adoption of automated planning.

To address these limitations, we present PDDL.jl, an extensible interpreter and compiler interface for fast and flexible AI planning. PDDL.jl exposes the semantics of planning domains through a common interface for executing actions, querying state variables, and other basic operations used within AI planning applications. PDDL.jl also supports the extension of PDDL semantics (e.g. to stochastic and continuous domains), domain abstraction for generalized heuristic search (via abstract interpretation), and domain compilation for efficient planning, enabling speed and flexibility for PDDL and its many descendants. Collectively, these features allow PDDL.jl to serve as a general high-performance platform for AI applications and research programs that leverage the integration of symbolic planning with other AI technologies, such as neuro-symbolic reinforcement learning, probabilistic programming, and Bayesian inverse planning for value learning and goal inference.

Thesis Supervisor: Vikash K. Mansinghka
Title: Principal Research Scientist, Brain and Cognitive Sciences

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor of Brain and Cognitive Sciences

Acknowledgments

This thesis would not have been possible if not for the particular intellectual environment I have had the good fortune to find myself in — one that sits at the intersection of AI, cognitive science, and programming languages. For that reason, I owe many thanks to my friends, peers, and mentors in the Probabilistic Computing Project and the Computational Cognitive Science group, as well as many overlapping circles. The seed for this thesis was arguably planted in the Goals, Plans, and Stories meetings convened by Josh Tenenbaum, one of my advisors, and Laura Schulz. In an early meeting, Kelsey Allen and João Loula gave a presentation on the original STRIPS problem solver, which caused my then nascent interest in PDDL and symbolic planning to blossom in inspiration and awe. Something about the symbolic approach to planning just felt so *right* and cognitively plausible, and as someone with a long-standing interest in modeling how people reason and act to bring about their values, I was hooked. Many other friends and peers were present at those meetings, including Sholei Croom, Nathalie Fernandez, Alex Lew, Cathy Wong, Sam Tenka, Tom Silver, Junyi Chu, and Zhutian Yang, and I would like to thank them, Laura, and Josh for their company and conversations through those times.

That same semester, I also took 6.820 Fundamentals of Program Analysis by Armando Solar-Lezama, along with many of the friends above. And while the coursework was frustrating at times, the knowledge I gained through it has been crucial for this thesis. Since then, I have learned a lot more about programming languages from fellow members of ProbComp: Alex for his work on MetaPPL, and for first teaching me meta-programming; McCoy Becker for his work on Jaynes and our work on Genify, which led me to a much deeper understanding of Julia; Feras Saad for his thought-provoking work on SPPL; Marco Cusumano-Towner for his paradigm-shifting design of Gen, which served as a tremendous inspiration for PDDL.jl; and of course my advisor and mentor Vikash Mansinghka, whose endless ideas, insightful mentorship, and enjoyment of getting into the nitty gritty have been invaluable for my growth as a researcher over the past two years — not to mention his continued

enthusiasm as I go off on substantial excursions from the land of Bayesian inference he is most familiar with. And while I mention ProbComp, I cannot afford not to thank Rachel Paiste and Amanda Brower. Without their check-ins, logistical, and administrative support, life would have been a lot harder as I navigated laptop failure and missed deadlines as I worked on this thesis. A shoutout too goes to Nishad Gothoskar, whose company at our weekly check-ins and frequent presence at lab has provided much-needed community and conversation in these pandemic times. And for the joy they have brought me as a mentor, I would like to thank my wonderful and talented mentees, Gloria Lin, Arwa Alanqary, Joie Le, and Jordyn Mann.

Finally, the trials and tribulations of the past few years would have been a lot more difficult if not for the support of many others outside of my life as a researcher. For their care, friendship, and camaraderie, I would like to thank Seran Gee, Alex Lew, Cathy Wong, Sholei Croom, Nathalie Fernandez, Nishad Gothoskar, Matt Bowers, Madeleine Laitz, Jules Drean, Florian Koehler, Willie Boag, Emma Batson, Jennifer Wang, Alison Biester, Alex Quinn, Elijah Gunther, Jacob Romm, Richard Baker, Andre Moura, Annabeth Leow, Jayashri Venketasubramanian, Kwok Yingchen, Lim Shu Ning, Theia Henderson, Turga Ganapathy, Jocelyn Wang, Eugenie Lai, and Tessa Green. And for their unconditional love and acceptance of who I am, have been, and will become, I would like to thank my dearest Mum and Dad, whom I have missed dearly since starting graduate school, and whose child I am proud to call myself.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

If we had better standards for evaluating AI, the field would progress faster.

John McCarthy

The nice thing about standards is that you have so many to choose from.

Andrew S. Tanenbaum

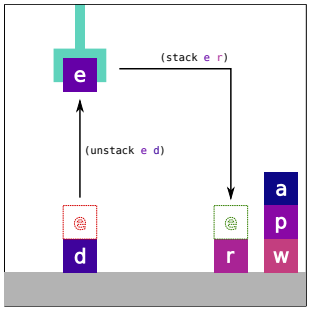
One of the earliest problems tackled by AI researchers in their quest to emulate human intelligence was symbolic planning and problem solving: deriving a series of steps to achieve a goal given a formal description of the world [1, 2]. This approach to autonomous decision-making remains important in many contemporary AI and robotics systems, which combine symbolic planning with stochastic world models [3], motion planning [4], Bayesian inference [5] and machine learning [6], thereby overcoming the early limitations of symbolic AI [7] while achieving much greater computational efficiency than brute-force deep learning algorithms [8, 9]. To enable the development and evaluation of such planning algorithms, the Planning Domain Definition Language (PDDL) emerged as a *de facto* standard for formally specifying planning problems and domains (Figure 1-1) as input to automated planners [10, 11]. Many extensions of PDDL have since been developed, enabling support for

```
(define (domain blocksworld)
  (:requirements :strips :typing :equality)
  (:types block)
  (:predicates (on ?x ?y - block) (ontable ?x - block)
               (clear ?x - block) (handempty) (holding ?x - block))
  (:action pick-up
   :parameters (?x - block)
   :precondition (and (clear ?x) (ontable ?x) (handempty))
   :effect (and (not (ontable ?x)) (not (clear ?x))
                (not (handempty) (holding ?x)))
  (:action put-down
   :parameters (?x - block)
   :precondition (holding ?x)
   :effect (and (not (holding ?x)) (clear ?x)
                (handempty) (ontable ?x)))
  (:action stack
   :parameters (?x ?y - block)
   :precondition (and (holding ?x) (clear ?y) (not (= ?x ?y)))
   :effect (and (not (holding ?x)) (not (clear ?y))
                (clear ?x) (handempty) (on ?x ?y)))
  (:action unstack
   :parameters (?x ?y - block)
   :precondition (and (on ?x ?y) (clear ?x) (handempty) (not (= ?x ?y)))
   :effect (and (holding ?x) (clear ?y) (not (clear ?x))
                (not (handempty)) (not (on ?x ?y))))
)
```

(a)

```
(define (problem bw-problem)
  (:domain blocksworld)
  (:objects a d e r p w - block)
  (:init (handempty)
    (ontable d) (on e d) (clear e)
    (ontable r) (clear r)
    (ontable w) (on p w) (on a p) (clear a))
  (:goal (and (ontable r) (on e r) (clear e)))
)
```

(b)

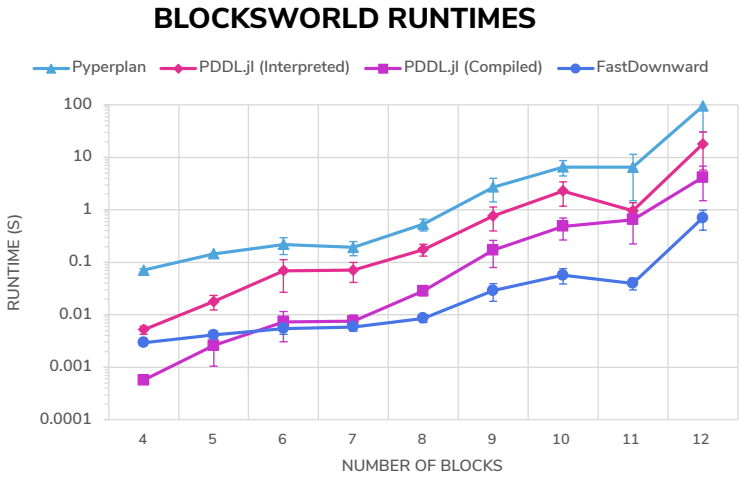


(c)

Figure 1-1: PDDL can be used to specify planning (a) domains and (b) problems in terms of symbolic relations and actions. (c) Blocksworld is a well-known example.

stochastic domains [12], continuous environments [13], and other real-world conditions [14]. PDDL and its descendants thus serve as key infrastructural components in AI planning and robotics research.

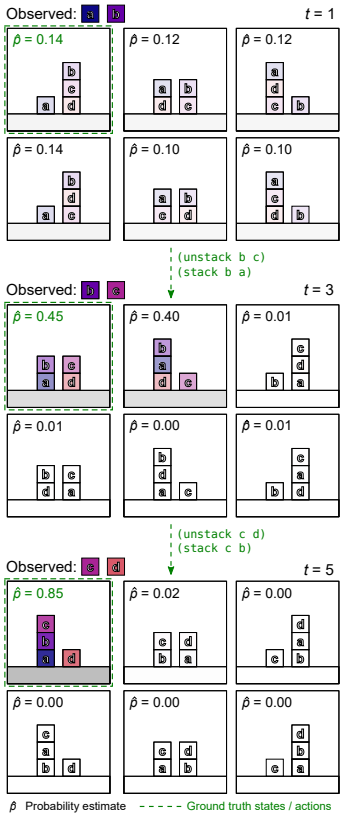
Despite the widespread of use of PDDL, however, most implementations of PDDL semantics are closely tied to particular planning systems or algorithms, each of which uses its own compiler for PDDL domain descriptions in order to optimize performance of the associated algorithm [15, 16, 17]. While this benefits the speed and efficiency of these planners, enabling them to excel at the semi-annual International Planning Competitions [18], this comes at the cost of code reuse, extensibility, and integration within larger AI systems. As a result, each proposed extension to PDDL has typically required a custom-built planner for that extension [19, 20, 21, 22], and the use of PDDL for broader applications such as plan recognition or task-and-motion-planning has required either specific translation techniques [23] or glue code [24]. This implementational barrier limits the uptake of symbolic planning by researchers outside of the planning community, and inhibits the integration of symbolic, probabilistic and neural approaches to AI that has been projected to play a key role in the next generation of autonomous systems [25, 26].



(a) Planning in PDDL.jl vs. Pyperplan vs. FastDownward

```
"Construct a symbolic MDP from a PDDL domain and problem."
function SymbolicMDP(domain::Domain, problem::Problem)
    state = PDDL.initstate(domain, problem)
    goal = PDDL.get_goal(problem)
    metric = PDDL.get_metric(problem)
    if metric !== nothing # Extract metric formula to minimize
        metric = metric.name == :minimize ?
            metric.args[1] : Compound(:-, metric.args)
    end
    return SymbolicMDP(domain, state, goal, metric)
end
```

(b) MDP and RL environments using PDDL.jl



(c) State estimation using PDDL.jl and Gen

Figure 1-2: A survey of PDDL.jl applications. PDDL.jl can be used: (a) To build planning algorithms 20 times faster than Pyperplan, and within an order of magnitude of FastDownward (Chapter 6.1); (b) As a simulator for reinforcement learning (RL) that works with existing RL packages (Chapter 6.2); (c) To perform state estimation from partial observations when combined with the Gen probabilistic programming system (Chapter 6.3).

To overcome this implementational barrier, this thesis presents PDDL.jl[1], an extensible interpreter and compiler interface designed for fast and flexible symbolic planning within a wide range of AI systems and applications (see Figure 1-2 for a survey). PDDL.jl exposes the semantics of PDDL planning domains through a common interface for executing actions, determining the set of applicable actions, querying state variables, and other basic planning operations, providing implementation-agnostic building blocks for planning algorithms and related AI systems. Through multiple implementations of this common interface, PDDL.jl also supports the extension of

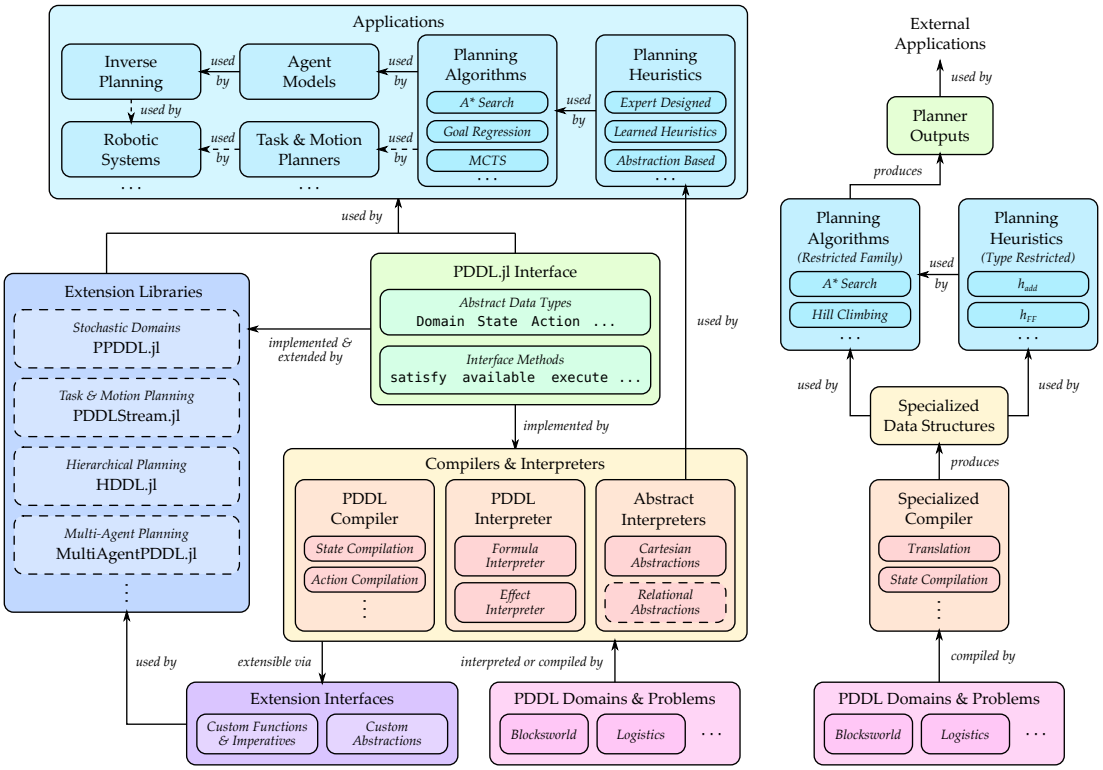
[1] Available at `https://github.com/JuliaPlanners/PDDL.jl`

PDDL semantics (via an extensible interpreter), domain abstraction for generalized heuristic search (via an abstract interpreter), and domain compilation for efficient planning (via a PDDL compiler), enabling speed and flexibility for applications that use PDDL or its many descendants. Collectively, these features allow PDDL.jl to serve as a platform for contemporary AI algorithms and research programs that leverage the integration of symbolic planning with other AI technologies, including task and motion planning [13, 27], neuro-symbolic reinforcement learning [28, 29], and Bayesian inverse planning for value learning and goal inference [30].

1.1 Overview

PDDL.jl differs from standard automated planning systems in that it is designed not only for speed and efficiency, but also extensibility and interoperability. This is due to the fact that the design target of PDDL.jl is an *interface*, not just a particular algorithm or application. Figure 1-3(a) provides an overview of the architecture of PDDL.jl and the ecosystem it enables, in comparison with the architecture of standard planning systems shown in Figure 1-3(b). Standard architectures are designed primarily for fast and efficient planning, accepting PDDL domain and problem files as inputs (Figure 1-3(b), pink), rapidly translating and compiling them (orange) to more efficient representations (yellow), running planning algorithms and heuristics (blue) over those representations, then producing symbolic plans and metadata as outputs (green). This architecture enables performance optimization over the entire pipeline, but limits interaction with external applications to just two channels: (i) receiving domains and problems as inputs; and (ii) providing plans as outputs.

In contrast, the core of PDDL.jl is its interface (Figure 1-3(a), green): a set of methods and abstract data types that expose the high-level functionality required to implement planning algorithms and applications. Centering PDDL.jl around its interface means that (i) multiple *implementations* of the interface can coexist (yellow), providing either speed, generality or specialized functionality depending on engineering needs; (ii) multiple *applications* (light blue) can use the interface to achieve



(a) PDDL.jl's architecture and ecosystem (b) Standard architecture

Figure 1-3: An overview of (a) the architecture and ecosystem of PDDL.jl, in contrast to (b) standard planning systems. Dashed lines surround libraries or features that are yet to be implemented, but are planned for the future.

tighter integration between symbolic planning and other AI components; and (iii) multiple *extensions* of PDDL are enabled by implementing and extending the interface through additional libraries (dark blue). By factoring out these components of traditional planning systems into separate software artifacts, PDDL.jl enables an ecosystem where implementations can evolve independently from applications (e.g. through future compiler improvements), applications can interoperate through a common interface (e.g. Bayesian agent models which interleave planning algorithms with plan execution [30]), and extensions can be flexibly composed (e.g. stochastic domains with task and motion planning [31]).

Given this interface-centered design, PDDL.jl itself does not include any applications or extensions, which are intended to be provided by separate libraries (e.g. the `SymbolicPlanners.jl` library of planning algorithms [32]). However, PDDL.jl does

include several built-in implementations of its interface: a standard interpreter, an abstract interpreter, and a compiler. Each of these implementations plays a different role in the context of a planning application and its development. The standard interpreter is designed to be easily extended, and also comes with the ease of debugging and inspection usually associated with interpreters. As such, it is ideal for checking correctness when specifying a new PDDL domain, or when implementing a planning algorithm or extension library. The abstract interpreter's primary intended use is to compute planning heuristics that rely upon domain relaxation or abstraction [33]. However, abstract interpreters have many other uses (e.g. program synthesis for generalized planning) which future applications could take advantage of. Finally, the PDDL.jl compiler enables efficient planning through just-in-time compilation of specialized state representations and action semantics. While compilation is less easy to extend or debug, it provides orders of magnitude speed-ups over interpretation, allowing PDDL.jl applications to scale to much larger problems.

In the following chapters, we describe the PDDL.jl interface and each of these implementations in detail. In Chapter 2, we introduce the abstract data types and interface methods that constitute the PDDL.jl interface, then provide examples of how it can be used by applications and extended by additional libraries. In Chapter 3, we describe the standard interpreter in the context of PDDL syntax and semantics, and how interpreter functionality can be easily extended. In Chapter 4, we introduce the abstract interpreter, and how it can be used to compute planning heuristics based upon domain abstraction. In Chapter 5, we describe the compiler and the optimizations it implements. Finally, in Chapter 6, we illustrate and evaluate several applications of PDDL.jl, including high-performance symbolic planning algorithms [32], environments for reinforcement learning [34], state estimation from partial observations, and goal inference via Bayesian inverse planning [30]. The thesis concludes with a discussion of future research that might be enabled by PDDL.jl.

Chapter 2

A General Software Interface for Symbolic Planning

Decades of AI research have led to the development of a wide variety of symbolic planning algorithms and associated software packages, each of them often tailored to support specific planning strategies (e.g. planning graph analysis in GraphPlan [35] vs. forward search in FastForward [16]) or certain classes of planning problems (e.g. numeric planning in MetricFF [19] or temporal planning in POPF [36]). Some of these software packages are more general than others; for example, both the Fast-Downward [17] and ENHSP [21, 22] planning systems support an array of algorithms and heuristics for classical and numeric planning respectively.

However, there exist very limited options when it comes to *software interfaces* for the integration of symbolic planning into larger systems. The most common approach is integration of existing planners at the invocation level (e.g. in ROSPlan [37]) which severely limits the degree of inter-operation, or source-code modification of slow but lightweight planner implementations such as Pyperplan [38]. More general interfaces are provided by the PDDL4J [39] and PPMaJaL [40] libraries, which define Java APIs for PDDL-based planners, heuristics, and various utilities. While these libraries are considerably more flexible than the aforementioned approaches, their design does not easily allow for deeper semantic extensions (e.g. to probabilistic or multiagent planning) or for alternative interpretation and compilation techniques.

To address these limitations, this chapter presents the PDDL.jl interface for symbolic planning, which is motivated by the following desiderata:

1. The interface should be sufficiently **general**, providing interface methods and data types for a variety of use cases. This includes classical planning algorithms, but also applications which operate over symbolic descriptions of the world (e.g. environment simulators or probabilistic state space models).

2. The interface should be designed for **extensibility**, allowing developers to support new classes of planning problems (e.g., continuous or stochastic environments) primarily by implementing new behavior for existing interface methods, with minimal extensions to the interface itself.

3. The interface should support **specialization** of its methods in a domain and problem-specific manner, so as to enable efficient planning through techniques such as domain compilation and finite state encodings.

These are achieved by a system of abstract data types and associated interface methods which mirror the formal semantics of symbolic planning problems, along with their implementation in the Julia programming language, enabling both extensibility and specialization through a combination of multiple dispatch and code generation. In the following sections, we first provide formal definitions of symbolic planning domains and problems, then introduce the system of abstract data types and interface methods for symbolic planning that are exposed to downstream applications.

2.1 Background on Symbolic Planning

Symbolic planning is a general term for approaches to automated planning and reasoning that describe the world and its dynamics in terms of high-level symbols, typically using first-order logic. PDDL is one way of representing such symbolic knowledge, but there are many other formalisms. Here we introduce general definitions of planning domains, problems, states, and actions (see Figure 2-1 for an overview). These definitions apply to PDDL and related languages such as STRIPS and ADL [41]. Variants

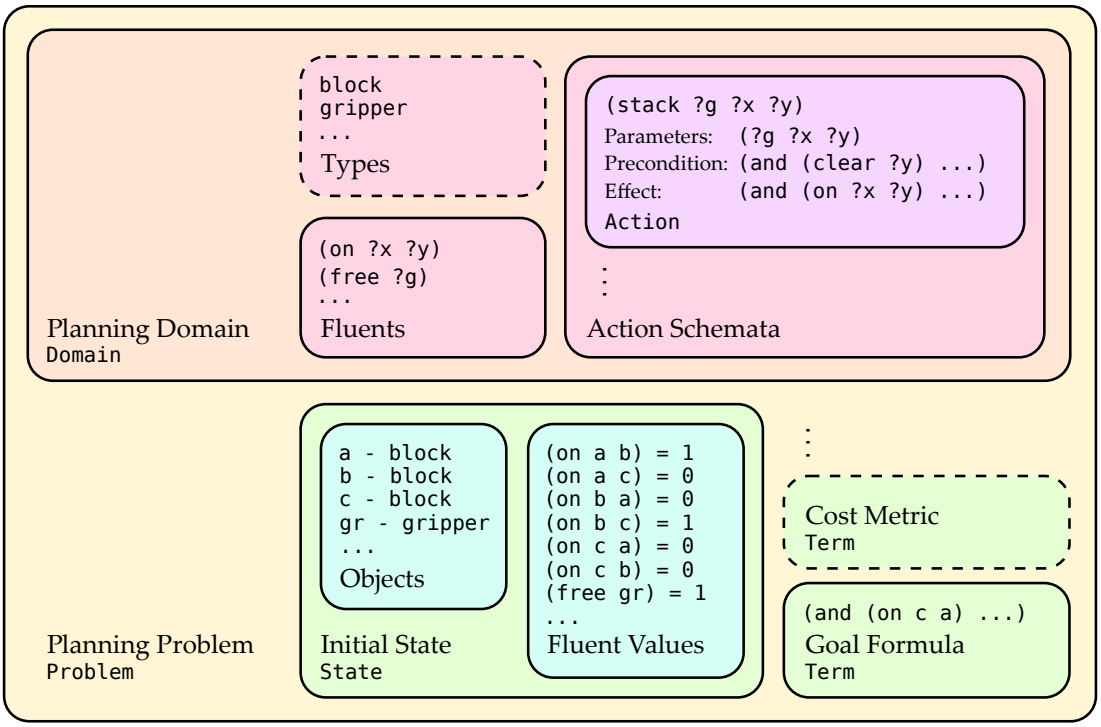


Figure 2-1: An overview of the relationships between planning domains, problems, states, actions, fluents, and logical terms in an example Blocksworld problem. Their corresponding abstract data types in PDDL.jl are annotated in `monospace`. Optional components are surrounded by dashed lines.

of PDDL also share these basic definitions, while extending them to incorporate more functionality and expressivity.

We first introduce the concept of fluents, which we use to define (relational) state variables which may change over time:

Definition 2.1.1 (Fluents)**.** A *fluent* F of arity n is a predicate (Boolean-valued) or function (non-Boolean) with n object arguments, which describes some property or relation over those arguments that may change over time. A *ground fluent* f is a fluent defined over particular set of objects. Arguments may optionally be typed.

Example. The fluent `(on ?x ?y)` is named `on`, has arity $n = 2$, and describes whether some object denoted by the variable `?x` is stacked on top of `?y`. The ground fluent `(on a b)` denotes that object `a` is stacked on top of object `b` when true.

States of the world can be symbolically described in terms of fluents and their valuations at a particular point in time:

Definition 2.1.2 (States)**.** Given a finite set of fluents $\mathcal{F}$, a *state* s is composed of a set of (optionally typed) objects $\mathcal{O} := \mathsf{objects}(s)$, and valuations of ground fluents $\mathcal{F}(\mathcal{O})$ defined over all objects in $\mathcal{O}$ of the appropriate types. Each ground fluent thus refers to a state variable. For a ground fluent $f \in \mathcal{F}(\mathcal{O})$, we use the notation $s[f] = v$ to denote that f has value v in state s.

Example. Given the fluent `(on ?x ?y)` and a state s with $\mathsf{objects}(s) = \{\mathtt{a}, \mathtt{b}\}$, the expression $s[\texttt{(on a b)}] = \texttt{true}$ means that object `a` is on top of `b` in state s.

The semantics of actions are specified via *action schemata*, which describe the preconditions and effects of actions in terms of the objects they operate over.

Definition 2.1.3 (Action Schemata)**.** An *action schema* A, also known as a *planning operator*, is a tuple $(\mathsf{params}(A), \mathsf{precond}(A), \mathsf{effect}(A))$, where:

- $\mathsf{params}(A)$ is a list of (optionally typed) parameter variables, which serve as placeholders for the objects that a concrete action operates over.
- $\mathsf{precond}(A)$ is a logical formula, defined using a set of fluents $\mathcal{F}$ over the variables in $\mathsf{params}(A)$, which specifies the preconditions under which an action $A(o)$ with object arguments o may be executed.
- $\mathsf{effect}(A)$ is a formula, defined over the variables in $\mathsf{params}(A)$, specifying the effects of an action $A(o)$ with arguments o. Effects can be assignments to fluents in a state, or more general expressions with conditions or probabilistic choices.

Some formalisms also support schemata for *durative actions*, i.e., actions whose preconditions or effects apply over time, with accordingly expanded definitions [42].

Definition 2.1.4 (Actions)**.** An *action* $a = A(o)$ is defined by an action schema A and arguments o, specifying the objects that a operates over.

An action is said to be *available* or *applicable* in a state s if $\mathsf{precond}(A)$ holds true in s when the variables in $\mathsf{params}(A)$ are substituted by their concrete values o. It is said to be *relevant* to a state s if some fluent valuation in s could possibly be achieved by some effect of a.

Having defined states and actions, we now define planning domains and problems.

Definition 2.1.5 (Planning Domains)**.** A *symbolic planning domain* $\mathcal{D}$ is a tuple $(\mathcal{T}, \mathcal{F}, \mathcal{A}, ...)$ where $\mathcal{T}$ is an (optional) set of object types, $\mathcal{F}$ a set of fluents that describe states, preconditions and effects, and $\mathcal{A}$ a set of action schemata defining the semantics of actions in $\mathcal{D}$. Some formalisms extend domains with other components, such as object constants or axioms for derived predicates [43].

A planning domain effectively specifies the (lifted) transition dynamics of a first-order symbolic model of the world. When all effects are deterministic, $\mathcal{D}$ defines a state transition system over all possible states $\mathcal{S}$ when paired with a set of objects $\mathcal{O}$ that states are defined over. When effects are stochastic, $\mathcal{D}$ defines a transition distribution $T(s'|s, a)$ over successor states s' conditioned on a state s and action a.

Definition 2.1.6 (Planning Problems)**.** A *symbolic planning problem* $\mathcal{P}$ is a tuple $(\mathcal{D}, s_0, g, ...)$, where $\mathcal{D} = (\mathcal{T}, \mathcal{F}, \mathcal{A}, ...)$ is a planning domain, s_0 is an initial state defined over objects with types in $\mathcal{T}$ and fluents in $\mathcal{F}$, and g is the goal to achieve, specified as a logical formula (e.g. a conjunction of fluents to be satisfied).

In addition to a goal formula, a planning problem may also include other specifications, such as a cost metric to minimize, and temporal constraints on the plan. The task of a planning algorithm is to find a sequence of actions from the initial state (a plan) or a mapping from states to actions (a policy) that leads to the problem specifications being satisfied or optimized.

2.2 Abstract Data Types for Symbolic Planning

To support reasoning over fluents, states, actions, domains and problems, we introduce a system of abstract data types (ADTs) in PDDL.jl to mirror these concepts:

Definition 2.2.1 (`Term`)**.** The `Term` ADT is used to represent fluents, object constants, free variables, logical formulae (e.g. in action preconditions), effect formulae (which may include imperative expressions), and ground actions. Every `Term` has a `name` property, indicating the object, variable, fluent, logical operation, imperative

operation, or action schema it refers to, as well as an `args` property, representing the (potentially empty) list of sub-terms it has as arguments.

Definition 2.2.2 (`State`)**.** The `State` ADT represents a symbolic state (Definition 2.1.2). Objects in a `State` s can be accessed with `get_objects(`s`::State)` and object types with `get_objtypes(`s`::State)`. The value $s[f]$ of a ground fluent f in s can be accessed with `get_fluent(`s`::State, `f`::Term)`, where f is represented as a `Term`. For concrete sub-types of `State`, `set_fluent!(`s`::State, `v`, `f`::Term)` allows the value v to be assigned to $s[f]$.

Definition 2.2.3 (`Action`)**.** The `Action` ADT represents an action schema (Definition 2.1.3). Each `Action` schema A has a name given by `get_name(`A`::Action)`, a list of parameter variables given by `get_argvars(`A`::Action)`, a precondition `Term` given by `get_precond(`A`::Action)`, and an effect `Term` given by `get_effect(`A`::Action)`.

Note that ground actions (Definition 2.1.4) are represented with the `Term` data type rather than the `Action` data type, because the `name` property of a `Term` is sufficient to identify an action schema in the context of a planning domain, and the `args` property can be used to represent action parameters.

Definition 2.2.4 (`Domain`)**.** The `Domain` ADT represents a planning domain (Definition 2.1.5). For a domain $\mathcal{D} = (\mathcal{T}, \mathcal{F}, \mathcal{A}, ...)$, `get_types(`$\mathcal{D}$`::Domain)` returns the set of types $\mathcal{T}$, `get_fluents(`$\mathcal{D}$`::Domain)` returns a map from fluent names to fluent terms in $\mathcal{F}$, and `get_actions(`$\mathcal{D}$`::Domain)` returns a map from action names to action schemata $\mathcal{A}$.

Definition 2.2.5 (`Problem`)**.** The `Problem` ADT represents a planning problem (Definition 2.1.6) sans the domain $\mathcal{D}$ it is defined over. For a `Problem` $\mathcal{P}$, the function `initstate(`$\mathcal{D}$`::Domain, `$\mathcal{P}$`::Problem)` returns the initial state s_0 specified by $\mathcal{P}$ as a `State`, and `get_goal(`$\mathcal{P}$`::Problem)` returns the goal specification g given by $\mathcal{P}$. Sub-types of the `Problem` ADT may provide additional methods for accessing other specifications (e.g. cost metrics).

Figure 2-1 provides an overview of these abstract data types and the entities they represent in an example Blocksworld problem.

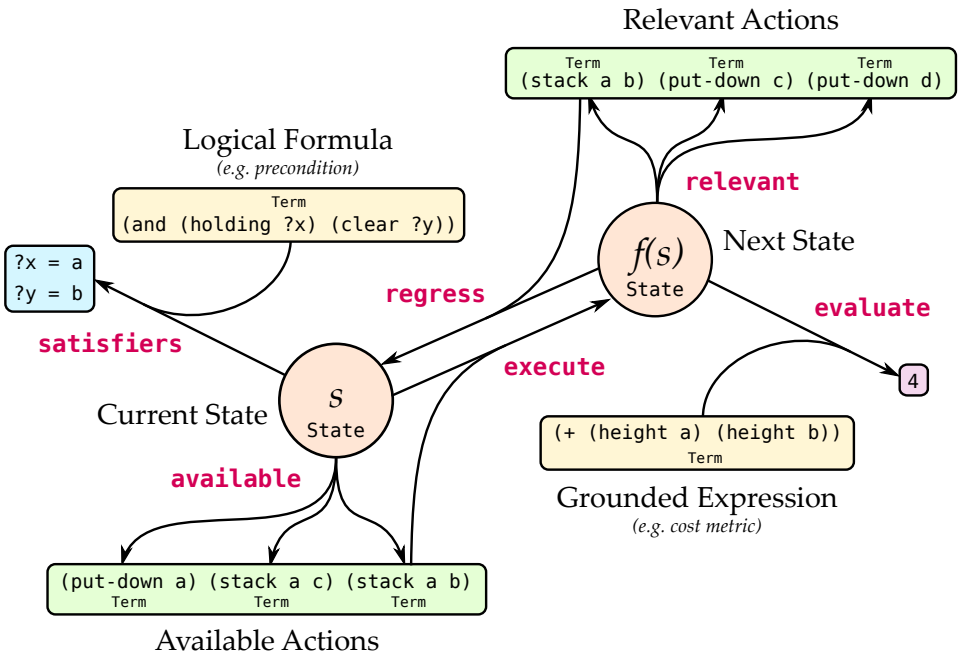


Figure 2-2: Schematic diagram of PDDL.jl interface methods (magenta, `monospace`): `satisfiers` returns all satisfying assignments of a logical formula in the context of a state; `evaluate` returns the value of a ground expression with respect to a state's fluents; `available` returns actions available in a state s; `execute` applies an action f to a state s; `relevant` returns all actions that could possibly lead to a state; `regress` executes an action f in reverse to get its pre-image.

2.3 Interface Methods for Symbolic Planning

We now introduce a set of interface methods that serve as basic operations in a wide variety of symbolic planning algorithms and applications, including forward state space search, backward-chaining search, partial order planning, and variations or combinations thereof. These methods are intended to be low-level enough such that planning algorithms can be expressed primarily in terms of the operations they represent, but high-level enough so as to abstract away from implementational details such as compiled state representations or optimizing domain transformations. An overview of most of these interface methods is presented in Figure 2-2.

Since the PDDL.jl interface is defined in Julia, we adopt Julia's syntax for declaring a method and its signature. For example, `foo(n::Int, x::Float64)` has the function name `foo`, arguments `n` and `x`, with types `Int` and `Float64` respectively.

If an argument is not annotated with a type, then it has no type restrictions. In addition, we make use of *multiple dispatch* in Julia, which allows a single function to have *multiple* methods, each with their own type signatures and behavior: the function `foo` might have an additional method `foo(n::Int)` that operates on only an `Int` argument. This feature allows us to group conceptually related functionality under the same name. It also allows us to declare an abstract interface by defining methods with ADTs as arguments, then implement the interface by defining methods of the same functions over concrete sub-types of those ADTs. This means that interfaces can be defined over a system of ADTs, not just a single ADT as is typical in object-oriented programming. We make use of this design pattern in defining the PDDL.jl interface.

2.3.1 Evaluating Formulae and Expressions

In contrast to planning approaches which assume less knowledge about the environment, the key distinguishing feature of symbolic planning is the ability to describe and determine whether certain facts about the world hold true (e.g. is the robot holding a block?), or evaluate numeric properties (e.g. the distance between two cities), where these queries are expressed in terms of first-order logic or a similar language. This allows planners to check whether logical specifications have been met, and for planning strategies which select actions that achieve sub-formulae of goal specifications. As such, we introduce the following methods which satisfy or evaluate first-order expressions in the context of a `State`:

Definition 2.3.1 (Formula Satisfaction)**.** Given a `term` representing a well-formed logical formula, or a collection of `terms` (treated as conjunctions of such formulae), the `satisfy` function returns whether they are satisfiable within a `domain` and `state`:

```
satisfy(domain::Domain, state::State, term::Term)
satisfy(domain::Domain, state::State, terms)
```

When a `term` has free variables, `satisfy` returns true as long as one satisfying assignment exists. A related function, `satisfiers`, returns a list of all satisfying as-

signments to such variables (a.k.a. substitutions), including the empty list when a variable-free formula is satisfied (the null value `nothing` is returned otherwise):

```
satisfiers(domain::Domain, state::State, term::Term)
```

Definition 2.3.2 (Term Evaluation)**.** Given a `term` representing a ground expression (i.e. one with no free variables), the `evaluate` function returns the value of that expression in the context of a `domain` and `state`:

```
evaluate(domain::Domain, state::State, term::Term)
```

For example, if `term` refers to a fluent, the value of the fluent is returned. Compound numeric expressions (e.g., the sum of two fluents) can also be evaluated. For ground logical formulae (i.e. sentences), `evaluate` is equivalent to `satisfy`.

2.3.2 Initialization and Transition Dynamics

As noted in Definitions 2.1.5 and 2.1.6, domains specify the transition dynamics of a first order symbolic model of the world, while problems specify the initial state and object set over which these dynamics are grounded. We thus introduce methods for constructing the initial state, and for simulating the transition dynamics:

Definition 2.3.3 (State Initialization)**.** Given a `domain` and `problem`, the `initstate` function returns the initial state, the type of which is concrete subtype of '`State`':

```
initstate(domain::Domain, problem::Problem)
```

The type of the returned state may vary depending on the subtype of the `domain` or `problem` provided. For example, in our compiler implementation (Chapter 5), providing a compiled `domain` as an argument leads `initstate` to return a compiled state representation.

Definition 2.3.4 (State Transition)**.** Given a `domain` and `state` and `action`, the `transition` function returns a successor state after applying `action` in `state`, including the effects of any exogenous activity (as supported by some temporal extensions to PDDL [44]) and random sampling (in the case of probabilistic effects [12]):

```
transition(domain::Domain, state::State, action::Term)
```

To support future multi-agent extensions of PDDL.jl, we also introduce a variant of `transition` which accepts a set of `actions` to be executed in parallel:

```
transition(domain::Domain, state::State, actions)
```

2.3.3 Forward Action Semantics

A widely-used strategy in symbolic planning is forward state space search, guided by a planning heuristic. These algorithms are built upon two basic operations to search forward in state space: querying the actions that are available in any given state, and executing an action to generate a successor state. This motivates the following interface methods:

Definition 2.3.5 (Action Availability)**.** Given a `domain`, `state`, `action` schema and action arguments, the `available` function returns whether the corresponding action is available in the specified `state` — i.e. its precondition is fulfilled. An `action` may alternatively be provided as a `Term` (e.g. `(stack a b)`):

```
available(domain::Domain, state::State, action::Action, args)
available(domain::Domain, state::State, action::Term)
```

When `available` is called without specifying an action, it returns an iterator over all actions available in the specified `state`, effectively encapsulating the logic for node expansion in a search algorithm:

```
available(domain::Domain, state::State)
```

Definition 2.3.6 (Action Execution)**.** Given a `domain`, `state`, `action` schema and action arguments, the `execute` function returns the result of applying the specified action to the `state`. An `action` may also be provided as a `Term`:

```
execute(domain::Domain, state::State, action::Action, args)
execute(domain::Domain, state::State, action::Term)
```

2.3.4 Inverse Semantics

Regression-based planners (e.g. the classical STRIPS algorithm [2]) make use of the fact that is possible to plan by working *backwards* from a goal, repeatedly selecting actions that are relevant to achieving a goal state or specification. This motivates the following interface methods for (i) constructing *abstract* states from goal specifications and (ii) exposing the *inverse* semantics of actions:

Definition 2.3.7 (Goal States)**.** In symbolic planning, a logical goal formula g effectively specifies the set of all concrete goal states where g holds true. We can represent this set of concrete states as an *abstract* state $\bar{s}$. In the special case where the goal g contains no disjunctions or functions, $\bar{s}$ can also be understood as a *partial* state that specifies the values of all predicates in g, and leaves all other predicates unspecified. To support regression search in this abstract space, PDDL.jl provides the `goalstate` method for constructing an abstract state from the goal specification of a problem:

```
goalstate(domain::Domain, problem::Problem)
```

As with `initstate`, the data type of the returned state $\bar{s}$ may depend on the `domain` or `problem` provided. For example, if the `domain` contains numeric fluents, `goalstate` may return an abstract state representation that specifies affine constraints on the numeric variables.

Definition 2.3.8 (Action Relevance)**.** Given a `domain`, `state`, `action` schema and action arguments, the `relevant` function returns whether the action is relevant to achieving the specified `state` — i.e., it achieves at least one predicate or numeric constraint in the `state`, and destroys none through deletion or modification. In the case where the action's effect reduces to a list of predicates to be added and a list to be deleted, this simplifies to checking that at least one added predicate is true in the state, and that none are deleted. An `action` may also be provided as a `Term`:

```
relevant(domain::Domain, state::State, action::Action, args)
relevant(domain::Domain, state::State, action::Term)
```

When `relevant` is called without specifying an action, it returns an iterator over all actions relevant to the specified `state`, encapsulating the logic for node expansion in a regression search algorithm:

```
relevant(domain::Domain, state::State)
```

Some planning algorithms also select actions based upon their relevance to particular sub-goals, expressed as logical `term`. As such, we provide a variant of `relevant` that returns an iterator over actions that may achieve a `term`:

```
relevant(domain::Domain, term::Term)
```

Definition 2.3.9 (Action Regression)**.** Given a `domain`, `state`, `action` schema and action arguments, the `regress` function executes the action in reverse, returning an iterator over (potentially abstract) states that constitutes the pre-image of the action with respect to the `state`. An `action` may also be provided as a `Term`:

```
regress(domain::Domain, state::State, action::Term)
regress(domain::Domain, state::State, action::Action, args)
```

Note that the `state` may be an abstract state that represents a set of concrete states, as returned by functions such as `goalstate`.

2.4 Using the Interface

A well-designed software interface should be both general enough to support the implementation of a wide variety of applications, but also simple enough that developers can easily make use of the interface. While evaluation of these design goals is best achieved through qualitative and quantitative user studies, we now provide two examples of how the PDDL.jl interface can be used in order to illustrate the both its generality and its simplicity.

Our first example is a forward breadth-first search algorithm, `bfs`, shown in Algorithm 1. The `bfs` algorithm accepts a `Domain` and `Problem`, then constructs the initial state with the `initstate` function. It also extracts the goal formula using `get_goal`.

Algorithm 1 Forward breadth-first search implemented using PDDL.jl.

```
function bfs(domain::Domain, problem::Problem)
    # Initialize state and extract goal
    state = initstate(domain, problem)
    goal = get_goal(problem)
    # Initialize search queue
    plan = []
    queue = [(state, plan)]
    while length(queue) > 0
        # Pop state and plan
        state, plan = pop!(queue)
        # Check if goal is satisfied
        if satisfy(domain, state, goal)
            # Return plan if goal is satisfied
            return plan
        end
        # Iterate over available actions and add successors to queue
        for action in available(domain, state)
            next_state = transition(domain, state, action)
            next_plan = [plan; action]
            pushfirst!(queue, (next_state, next_plan))
        end
    end
    # Return nothing upon failure
    return nothing
end
```

The algorithm then performs breadth-first search of the state space, popping a state and corresponding plan off the search queue at each iteration, and checking if the state satisfies the goal with the `satisfy` function. If the goal is satisfied, the plan is returned. If not, the state is expanded by iterating over each available action with `available(domain, state)` and constructing the successor state by applying the effects of that action using the `transition` function. Each successor state and corresponding plan is added to queue, and the search continues until either the queue is exhausted, or the `goal` is satisfied.

Our second example, `regression`, is similar to the original STRIPS algorithm for planning as theorem-proving [2], but expressed as backward search in the space of abstract states representing intermediate goals [15]. The algorithm first constructs an initial state (using `initstate`) and abstract goal state (using `goalstate`) from the domain and problem. It then performs breadth-first search from the *goal* state, iterating over actions that are *relevant* to achieving the goal via `relevant(domain,`

Algorithm 2 Regression planner implemented using PDDL.jl.

```
function regression(domain::Domain, problem::Problem)
    # Construct initial state and goal state
    init_state = initstate(domain, problem)
    state = goalstate(domain, problem)
    # Initialize search queue
    plan = []
    queue = [(state, plan)]
    while length(queue) > 0
        # Pop state and plan
        state, plan = pop!(queue)
        # Return plan if initial state implies the current abstract state
        if all(evaluate(domain, init_state, fluent) == val
               for (fluent, val) in get_fluents(state))
            return plan
        end
        # Iterate over relevant actions and add pre-image states to queue
        for action in relevant(domain, state)
            next_plan = [action; plan]
            for s in regress(domain, state, action)
                pushfirst!(queue, (s, next_plan))
            end
        end
    end
    # Return nothing upon failure
    return nothing
end
```

`state)`, then computing the preimage induced by each action using `regress` and adding the resulting states to the queue. The search terminates when the initial state is found to be in the preimage of some action, i.e., all fluents that are true in the preimage are also true in the initial state.

Variants of Algorithm 1 and 2 can be implemented with minor modifications to the code. For example, *best*-first search can be implemented by using a priority queue and incorporating a search heuristic. Different choices for how nodes are expanded and added to the search frontier lead to other variants such as hill-climbing search. Importantly, all domain and implementation specific details are encapsulated by the PDDL.jl interface, allowing the same algorithm to operate across multiple domains, and even multiple representations of the same domain (e.g. interpreted vs. compiled).

2.5 Extending the Interface

The PDDL.jl interface is designed so that it can be easily extended to support planning formalisms beyond the discrete and deterministic domains that PDDL was initially intended for. This extensibility is primarily due to the fact that functions in Julia are *open*: Julia libraries that use PDDL.jl can also extend the PDDL.jl interface by defining new methods for functions declared in PDDL.jl. In this section, we provide high-level sketches of how PDDL.jl might be extended to support such formalisms, including temporal planning, stochastic domains, task and motion planning, hierarchical planning, and multi-agent planning, thereby demonstrating the generality and coverage of the PDDL.jl interface. In the future, some of these extensions may be implemented in extension libraries for PDDL.jl (see Figure 1-3, dark blue).

2.5.1 Temporal Planning

Temporal planning involves reasoning over *durative actions* with effects or preconditions that apply over that time (introduced in PDDL 2 [45]), and possibly *events* and *processes* which are triggered once their preconditions hold true (introduced in PDDL+ [44]). While PDDL.jl does not support temporal reasoning by default, such reasoning can implemented by (i) introducing `DurativeAction`, `Event` and `Process` as ADTs; (ii) introducing concrete subtypes of the `State` ADT that store temporal information; (iii) defining new methods for `execute`, `transition` and `regress` which manipulate temporal information; and in the case of `transition`, (iv) ensuring that events and processes are automatically triggered.

One approach to tracking temporal information would be to use a `TemporalState` data type that stores not just the values of fluents, but also all on-going actions, a set of temporal constraints between action execution times, and a summarized *history* of recent changes, tagging each fluent with the action it was most recently changed by and the timestamp of that change. This augmented state representation is sufficient for temporal planning algorithms such as POPF [36] to ensure that the execution of new actions is temporally consistent with previous or ongoing actions.

2.5.2 Stochastic Domains

Actions in PDDL can be augmented with probabilistic effects (introduced in Probabilistic PDDL [12]), allowing this augmented form of PDDL to represent stochastic domains and first order Markov decision processes (FOMDPs) [46]. In order to support planning over such stochastic domains, we can extend the semantics of `execute` and `transition` so that they perform *sampling* of successor states for actions with probabilistic effects. Section 3.4 discusses how this can be implemented by extending PDDL.jl's built-in interpreter.

This minimal change to the PDDL.jl interface is sufficient for MDP solvers that only require sampling of future states, such as model-free reinforcement learning or Monte Carlo Tree Search (MCTS). To support other MDP solvers, we might introduce `transitdist` as a new interface method which returns an explicit distribution over successor states. This distribution could then be iterated over in algorithms that combine dynamic programming with heuristic search, such as Real-Time Dynamic Programming [3] or LAO* [47].

2.5.3 Task and Motion Planning

Planning in real-world settings (e.g. in robotics) requires reasoning over both symbolic object properties and relations as well as continuous quantities such as object poses, joint angles and motion trajectories. This hybrid planning formalism is called task and motion planning (TAMP) [27], as it requires the integration of symbolic planning (i.e. task planning) with motion planning.

The primary challenge introduced by TAMP problems is the infinite set of ground actions introduced by needing to select particular poses or continuous trajectories as action parameters. In order to overcome this challenge, many TAMP approaches augment planning systems with the ability to sample continuous action parameters using externally specified procedures (e.g. streams in the PDDLStream framework [13]). One approach to using such samplers with PDDL.jl is to allow samplers to be attached to planning domains. For such domains, action iterators such as `available(domain,`

`state)` and `relevant(domain, state)` would then iterate infinitely over actions with sampled continuous parameters. This integration strategy would allow planning algorithms that do not fully enumerate over actions (e.g. enforced hill climbing [48] or MCTS) to operate over continuous domains.

Another challenge introduced by TAMP settings is the need to verify continuous constraints or evaluate continuous effects, which may involve arbitrary non-linear functions. Some TAMP approaches circumvent this problem by delegating all constraint satisfaction to the sampling interface [13]. An alternative approach is to introduce *semantic attachments*: external function calls which are attached to PDDL domains to verify constraints and compute effects [24]. PDDL.jl provides built-in support for attaching custom functions to domains, as described in Section 3.2.

2.5.4 Hierarchical Planning

Hierarchical planning is an approach to planning where tasks are achieved by recursively decomposing them into lower-level tasks until reaching the level of primitive actions. Hierarchical task networks (HTNs) are a widely used formalism for this approach to planning [49], and can be expressed in extensions to PDDL such as the Hierarchical Domain Description Language (HDDL) [50]. The key additional operation performed in HTN planning is the decomposition of a higher level task into a network of subtasks. To that end, we can introduce an interface method, `decompose`, which accepts a non-primitive task and a decomposition method as inputs, and returns the network of subtasks as outputs.

Since primitive tasks are equivalent to (ground) actions, we can continue using the `available` and `execute` methods for such tasks. In addition, we can extend the semantics of `available` and `execute` to task networks by appropriately composing the preconditions and effects of the component tasks. The `available` method should check that all sub-tasks are primitive actions, that the preconditions of the first subtask hold, and that the effects of each sub-task are consistent with the preconditions of subsequent sub-task. The `execute` method should execute each component task in the (partial) order specified by the task network.

2.5.5 Multi-Agent Planning

Multi-agent planning extends symbolic planning to the setting where multiple agents in a single environment must either co-operate or compete to achieve some goal. In co-operative contexts, planning can be centralized, with a single controller coordinating multiple agents, or decentralized, with each agent forming its own plan. Multi-agent PDDL (MA-PDDL) [51] supports these extensions to PDDL by allowing the effects of actions to conditionally depend upon other actions being jointly executed, and by introducing an agent field for actions.

PDDL.jl can be extended to support these joint action semantics by defining methods for `available`, `execute`, `relevant`, `regress`, and `transition` that accept multiple actions as inputs. For example, `available` would return true only for action sets that are jointly applicable, while `execute` would take care of any interactions between simultaneously executed actions. These methods could then directly be used by centralized multi-agent planners to search for joint plans that achieve a shared goal. In decentralized contexts, a planner could instead sample or enumerate over the possible actions of cooperating or competing agents, and use them to form joint action sets that can be passed to `execute` for the purposes of planning ahead.

. . .

Having introduced the PDDL.jl interface for symbolic planning and a roadmap for future extensions, in the next few chapters we present several implementations of the interface built into PDDL.jl: a standard PDDL interpreter (Chapter 3), an abstract PDDL interpreter (Chapter 4), and a PDDL compiler (Chapter 5).

Chapter 3

An Extensible Interpreter for PDDL Domain Semantics

In the previous chapter, we introduced a general interface for symbolic planning applications. While this interface was designed with PDDL in mind, it is not strictly tied to any planning description language as long as that language shares the basic concepts introduced in Section 2.1. In order to support planning over PDDL domains and problems specifically, we need to implement the semantics of PDDL and expose them through the PDDL.jl interface.

We describe a straightforward approach to doing so in this chapter, namely, implementing a PDDL interpreter. Although interpreters are usually slow relative to compiled implementations of a language, they benefit from ease of implementation and increased transparency for debugging and development purposes. Indeed, these two advantages motivate us to design an interpreter that is *extensible*, allowing new language features and constructs to be added with relative ease. This is useful because of the many extensions and variants of PDDL which introduce new types of preconditions or effects. While it is possible to support an extension of PDDL by writing an entirely separate implementation of the PDDL.jl interface, designing the built-in interpreter to be extensible means that extension developers incur substantially less implementation cost, avoiding code duplication while benefiting from the ease of debugging new language features.

We focus here on interpreting logical formulae (used to express preconditions and goals) and effects, as these form the core semantics of PDDL — actions are then defined in terms of their preconditions and effects, domains by a set of action schemata along with sets of recognized types and fluents, and problems by goal formulae paired with assignments to fluents. Hence, we omit discussion of the syntax of actions, domains, and problems in PDDL (a full grammar is given in [52], and an example in Figure 1-1), and refer readers to Section 2.1 for formal definitions. We also restrict our scope to the non-temporal fragment of PDDL, as temporal planning is sufficiently different that we believe it warrants separate treatment (see Section 2.5.1). In the following sections, we describe the interpretation of precondition and goal formulae, the interpretation of effects, how both can be extended, and how action semantics can be interpreted in reverse for the purposes of regression search.

3.1 Interpreting Preconditions and Goals

Preconditions and goals in PDDL are expressed as first-order logical formulae, with a syntax and semantics essentially equivalent to standard first order logic with a few modifications: the addition of object types, and the restriction that predicates and functions are only defined over object constants (relaxed in some variants of PDDL). Full treatments of first order logic can be found in standard textbooks [53]. Here we present the most relevant aspects to interpreting preconditions and goals in PDDL.

PDDL formulae are written using a Lisp-style syntax, the grammar for which is shown in Figure 3-1. Apart from the standard logical connectives, PDDL also supports *typed* quantifiers over objects, which is the only major difference from standard first-order logic. For example, the formula `(forall (?b - block) (on-table ?b))` expresses that all objects of type `block` are located on the table. Variables are denoted by a `?` in front of their names, and other atomic terms are either objects, zero-arity functions, or numeric constants. PDDL.jl parses these expressions to syntax trees of abstract type `Term`, with `Compound` representing compound terms, `Var` representing variables, and `Const` representing (non-variable) atomic terms.

⟨*formula*⟩	::=	`(and` ⟨*formula*⟩* `)`	
	|	`(or` ⟨*formula*⟩* `)`	
	|	`(imply` ⟨*formula*⟩ ⟨*formula*⟩ `)`	
	|	`(not` ⟨*formula*⟩ `)`	
	|	`(forall (`⟨*typed-vars*⟩`)` ⟨*formula*⟩ `)`	
	|	`(exists (`⟨*typed-vars*⟩`)` ⟨*formula*⟩ `)`	
	|	`(`C ⟨*f-expr*⟩ ⟨*f-expr*⟩ `)`	where C is a binary comparison
	|	`(`P ⟨*atom*⟩* `)`	where P is a predicate symbol
⟨*typed-vars*⟩	::=	⟨*var*⟩*	
	|	⟨*var*⟩$^+$ `-` t ⟨*typed-vars*⟩	where t is a type name
⟨*f-expr*⟩	::=	`(`B ⟨*f-expr*⟩ ⟨*f-expr*⟩ `)`	where B is a binary operator
	|	`(`F ⟨*atom*⟩* `)`	where F is a function symbol
	|	x	where $x \in \mathbb{R}$ is a numeric constant
⟨*atom*⟩	::=	⟨*var*⟩	
	|	o	where o is an object name
⟨*var*⟩	::=	`?`v	where v is a variable name

Figure 3-1: Syntax for logical formulae in PDDL

A denotational semantics for PDDL formulae is presented in Figure 3-2, showing the mapping $\mathbb{F}[\![\cdot]\!]$ between PDDL and first-order logical expressions in a language $\mathcal{L}$, as written in standard mathematical notation. Again, the only major difference from standard first-order logic is that typed quantifiers introduce `typeof` predicates into the formulae they quantify over.

To specify when a formula $\phi \in \mathcal{L}$ is satisfied in a state s of domain $\mathcal{D}$, which we denote by $(s, \mathcal{D}) \models \phi$, we give a model-theoretic formalization: A state s in $\mathcal{D}$ represents a *structure* $\mathcal{M} = (\mathcal{U}, \sigma, I)$, where $\mathcal{U}$ is the *universe* of entities over which predicates and functions are defined; σ is the *signature* which contains predicate names, function names and their corresponding arities; and I is the interpretation function, an assignment of values to predicates and functions. For non-numeric domains, the universe $\mathcal{U} = \mathsf{objects}(s) \cup \mathcal{T}$ comprises the objects in s and the types $\mathcal{T}$ in $\mathcal{D}$. With numeric fluents, we also include the real numbers $\mathbb{R}$. The signature σ is specified by the fluents $\mathcal{F}$ of $\mathcal{D}$, with the addition of the `typeof` predicate. Finally, the interpretation I is specified by the values of all ground fluents f in state s, such that $I(f) = s[f]$. We also include the standard interpretations for arithmetic operators ($+, >, =$, etc.).

$\mathbb{F}[\![expr]\!] : \mathcal{L}$

$\mathbb{F}[\![\texttt{(and } f_1 \text{ ... } f_n\texttt{)}]\!]$	$=$	$\bigwedge_{i=1}^{n} \mathbb{F}[\![f_i]\!]$
$\mathbb{F}[\![\texttt{(or } f_1 \text{ ... } f_n\texttt{)}]\!]$	$=$	$\bigvee_{i=1}^{n} \mathbb{F}[\![f_i]\!]$
$\mathbb{F}[\![\texttt{(imply } f_1\ f_2\texttt{)}]\!]$	$=$	$\neg\mathbb{F}[\![f_1]\!] \vee \mathbb{F}[\![f_2]\!]$
$\mathbb{F}[\![\texttt{(not } f\texttt{)}]\!]$	$=$	$\neg\mathbb{F}[\![f]\!]$
$\mathbb{F}[\![\texttt{(forall } v_1 \texttt{ - } t_1 \text{ ... } v_k \texttt{ - } t_k\ f\texttt{)}]\!]$	$=$	$\forall v_1, ..., \forall v_k\ (\bigwedge_{i=1}^{k} \texttt{typeof}(v_i, t_i) \wedge \mathbb{F}[\![f]\!])$
$\mathbb{F}[\![\texttt{(exists } v_1 \texttt{ - } t_1 \text{ ... } v_k \texttt{ - } t_k\ f\texttt{)}]\!]$	$=$	$\exists v_1, ..., \exists v_k\ (\bigwedge_{i=1}^{k} \texttt{typeof}(v_i, t_i) \wedge \mathbb{F}[\![f]\!])$
$\mathbb{F}[\![\texttt{(}C\ f_1\ f_2\texttt{)}]\!]$	$=$	$\mathbb{F}[\![f_1]\!]\ C\ \mathbb{F}[\![f_2]\!]$ where $C \in \{=, <, \leq, >, \geq\}$
$\mathbb{F}[\![\texttt{(}B\ f_1\ f_2\texttt{)}]\!]$	$=$	$\mathbb{F}[\![f_1]\!]\ B\ \mathbb{F}[\![f_2]\!]$ where $B \in \{+, -, *, /\}$
$\mathbb{F}[\![\texttt{(}P\ a_1 \text{ ...} a_n\texttt{)}]\!]$	$=$	$P(\mathbb{F}[\![a_1]\!], ..., \mathbb{F}[\![a_n]\!])$ where P is a predicate symbol
$\mathbb{F}[\![\texttt{(}F\ a_1 \text{ ...} a_n\texttt{)}]\!]$	$=$	$F(\mathbb{F}[\![a_1]\!], ..., \mathbb{F}[\![a_n]\!])$ where F is a function symbol
$\mathbb{F}[\![\texttt{?}v]\!]$	$=$	v a variable
$\mathbb{F}[\![x]\!]$	$=$	x a numeric constant
$\mathbb{F}[\![o]\!]$	$=$	o an object constant

Figure 3-2: Denotational semantics of logical terms and formulae in PDDL.

We can now state that a ground formula ϕ is *satisfied* in a state s of domain $\mathcal{D}$ when their corresponding structure $\mathcal{M}$ satisfies ϕ, denoted $\mathcal{M} \models \phi$. For a formula ψ with free variables $v_1, ...v_k$, we say that ψ is *satisfiable* in a state s of domain $\mathcal{D}$ when $\mathcal{M} \models \exists v_1, ..., v_k\ \psi$, i.e., there exists some assignment μ to variables $v_1, ..., v_k$ such that ψ is satisfied. In the case where a domain $\mathcal{D}$ also specifies a set of axioms $\{a_1, ..., a_n\}$ for deriving certain predicates from others, we require that the axioms are simultaneously satisfied for a formula ϕ to be satisfied, i.e., $\mathcal{M} \models \phi \wedge a_1 \wedge ... \wedge a_n$. This formalizes the semantics of the `satisfy` interface method (Definition 2.3.1). The `satisfiers` interface method simply enumerates over all satisfying variable assignments.

In order to implement these satisfiability semantics, we can manually walk the syntax tree for a ground formula ϕ, checking whether its predicates are satisfied by state s, and whether all comparisons between functional terms hold true. For a formula ψ with free variables however, we either need to enumerate over all potential variable assignments (i.e. all combinations of all objects in s), or use a dedicated solver for first order logic. While many such solvers exist (e.g. standard implementations of Prolog, or the Z3 automated theorem prover), we opt to use Julog.jl, a Julia implementation of Prolog-style logical programming which was developed by

the author as a precursor to this thesis [54]. The `satisfy` and `satisfiers` interface methods are thus easily implemented by concatenating all state fluents and domain axioms into a database of first-order clauses, then passing that database to Julog along with the query formula. Julog then returns all satisfying assignments to any free variables in the query. The tight integration offered by Julog also means that we can pass in custom function definitions, which we can use to specify the values of numeric fluents. It also allows us to easily extend the semantics of PDDL formulae by providing external implementations for custom functions.

3.2 Extending Formulae with Custom Functions

Most real-world planning problems cannot be fully specified with the logical and linear arithmetic primitives supported by standard PDDL. For example, sorting objects may require reasoning about lists or sets, while motion planning requires reasoning about continuous physics and 3D geometry. To meet this challenge, numerous approaches have been proposed for extending the semantics of PDDL, such as *semantic attachments* that link predicates in PDDL with custom external implementations [24], or the *Planning Modulo Theories* framework for supporting new (mathematical) theories [33]. PDDL.jl enables such extensions by supporting the evaluation of *custom functions* within precondition and goal formulae.

As an example, consider the PDDL domain shown in Figure 3-3(a), which attempts to model projectile motion of a ball that can be thrown at or over a wall. There are two actions, `pick` and `throw`, the latter of which takes the angle `theta` as a continuous input parameter, and has the precondition that the `height` of the ball must be sufficiently large in order for the ball to reach the other side of the wall. However, specifying the semantics of `height` as a function of the launch angle and throw speed requires trigonometry, which PDDL does not natively support. With PDDL.jl, we can easily write these functions in Julia, then attach them to the domain, as shown in 3-3(b). PDDL.jl's interpreter will then evaluate these functions in the course of determining if an action is available or if a goal is achieved.

```
(define (domain ball-throw)
  (:requirements :fluents)
  (:predicates (has-ball ?ball) (on-floor ?ball))
  (:functions (throw-speed) (wall-height) (wall-dist) (loc ?ball)
              (range ?speed ?theta) (height ?speed ?theta ?dist))
  (:action pick
   :parameters (?ball)
   :precondition (and (on-floor ?ball))
   :effect (and (has-ball ?ball) (not (on-floor ?ball)))
  )
  (:action throw
   :parameters (?ball ?theta)
   :precondition (and (has-ball ?ball)
                      (> (height throw-speed ?theta wall-dist) (wall-height)))
   :effect (increase (loc ?ball) (range throw-speed ?theta))
  )
)
```

(a) PDDL domain specification

```
# Load domain
domain = load_domain("ball-throw.pddl")
# Define functions
function throw_range(v, theta)
    return v^2*sin(theta) / 9.81
end
function throw_height(v, theta, x)
    return tan(theta)*x - 9.81*x^2 / (2*v^2 *cos(theta)^2)
end
# Attach functions to domain
PDDL.attach!(domain, "range", throw_range)
PDDL.attach!(domain, "height", throw_height)
```

(b) Custom functions attached to the domain using PDDL.jl

Figure 3-3: A ball throwing domain in PDDL, with custom functions specified in Julia via PDDL.jl to encode the physics of 2D projectile motion.

This functionality can also be employed to support custom *theories* for symbolic planning, as introduced by the Planning Modulo Theories [33]. Rather than attaching custom functions to a particular planning domain, we can write an extension library for PDDL.jl that extends the semantics of its interpreter at a global level, allowing it to reason over mathematical objects such as sets and arrays. Figure 3-4 shows an example of how this can done with minimal implementation effort. In Figure 3-4(a), we show declarations of set-theoretic operations in the Module Definition Description Language (MDDL) proposed by [33] for describing new theories in a modular

```
(define (module set)
    (:type set of a')
    (:functions
        (construct-set ?x+ - a') - set of a'
        (empty-set) - set of a'
        (cardinality ?s - set of a') - integer
        (member ?s - set of a' ?x - a') - boolean
        (subset ?x - set of a' ?y - set of a') - boolean
        (union ?x - set of a' ?y - set of a') - set of a'
        (intersect ?x - set of a' ?y - set of a') - set of a'
        (difference ?x - set of a' ?y - set of a') - set of a'
        (add-element ?s - set of a' ?x - a') - set of a'
        (rem-element ?s - set of a' ?x - a') - set of a'
))
```

(a) Declarations of set-theoretic operations in MDDL [33]

```
# Register implementations of set operations
using PDDL
PDDL.register!(:function, "construct-set", (xs...) -> Set(xs))
PDDL.register!(:function, "empty-set", () -> Set())
PDDL.register!(:function, "cardinality", s::Set -> length(s))
PDDL.register!(:function, "member", (s::Set, x) -> in(x, s))
PDDL.register!(:function, "subset", (x::Set, y::Set) -> issubset(x, y))
PDDL.register!(:function, "union", (x::Set, y::Set) -> union(x, y))
PDDL.register!(:function, "intersect", (x::Set, y::Set) -> intersect(x, y))
PDDL.register!(:function, "difference", (x::Set, y::Set) -> setdiff(x, y))
PDDL.register!(:function, "add-element", (s::Set, x) -> union(s, x))
PDDL.register!(:function, "rem-element", (s::Set, x) -> setdiff(s, x))
```

(b) Set operations defined through PDDL.jl.

Figure 3-4: PDDL.jl can be extended to support reasoning over new data types and mathematical theories, such as the set operations declared in the Module Definition Description Language (MDDL) of [33]. These operations (a) can be implemented by mapping them to (b) corresponding implementations in Julia.

manner. In Figure 3-4(b), we show a corresponding implementation of these operations, facilitated by PDDL.jl. These operations will automatically be understood by the built-in PDDL interpreter when evaluating preconditions. They may also be used by PDDL.jl's compiler (Chapter 5), which can be configured to look up these implementations and integrate them into compiled code.

These two examples demonstrate the generality and reach of PDDL.jl's built-in interpreter in its ability to support new evaluation and satisfiability semantics for a broad range of theories and domains. However, one limitation of this functionality is that does not automatically inform PDDL.jl how to find satisfying assignments to

free variables in logical formulae when those variables are no longer restricted to a finite domain of objects. This is especially relevant to finding satisfying values for continuous action parameters, and hence determining the set of applicable actions within a state. The interpreter can determine whether an action is applicable once all parameters are specified, but cannot generate available groundings of an action schema when some of its parameters are not objects.

Fortunately, the extensibility of PDDL.jl is not limited to its interpreter. Users may also supply custom implementations of the `available(domain, state)` method, augmenting it with sampling semantics so that it generates a potentially infinite stream of actions (see Section 2.5.3). Such extensions would then allow planning algorithms that do not enumerate all actions to continue operating over the new (potentially infinite) domain in searching for a goal.

3.3 Interpreting Effect Expressions

⟨*effect*⟩	::=	`(`P ⟨*atom*⟩`*)`	where P is a predicate symbol
	|	`(not (`P ⟨*atom*⟩`*))`	where P is a predicate symbol
	|	`(`⟨*assign-op*⟩ `(`F ⟨*atom*⟩`*)` ⟨*f-expr*⟩`)`	where F is a function symbol
	|	`(and` ⟨*effect*⟩`*)`	
	|	`(when` ⟨*formula*⟩ ⟨*effect*⟩`)`	
	|	`(forall (`⟨*typed-vars*⟩`)` ⟨*effect*⟩`)`	
⟨*assign-op*⟩	::=	`assign` | `increase` | `decrease` | `scale-up` | `scale-down`	

Figure 3-5: Syntax for effect expressions in PDDL.

Similar to preconditions and goals, effects in PDDL are written using a Lisp-style syntax, the grammar for which is shown in Figure 3-5. The corresponding semantics are shown in Figure 3-6 using the semantic function $\mathbb{C}[\![\cdot]\!]$ (where $\mathbb{C}$ stands for command) that maps an input state $s \in \mathcal{S}$ to an output state $s' \in \mathcal{S}$. Traditionally, symbolic planning systems only supported two kinds of effects, *additions* and *deletions* of Boolean predicate. PDDL supports this by treating all (non-negated) terminal predicates in an effect syntax tree as additions, and all negated predicates as deletions. PDDL also allows for conditional effects using the `when` keyword, and

$\mathbb{C}[\![expr]\!] : \mathcal{S} \to \mathcal{S}$

$$
\begin{aligned}
\mathbb{C}[\![\texttt{(}P\ a_1\ ...a_n\texttt{)}]\!]s &= s[P(a_1,...,a_n) \mapsto \texttt{true}] \\
\mathbb{C}[\![\texttt{(not (}P\ a_1\ ...a_n\texttt{))}]\!]s &= s[P(a_1,...,a_n) \mapsto \texttt{false}] \\
\mathbb{C}[\![\texttt{(assign }f_1\ f_2\texttt{)}]\!]s &= s[\mathbb{F}[\![f_1]\!] \mapsto v],\ v = s[\mathbb{F}[\![f_2]\!]] \\
\mathbb{C}[\![\texttt{(increase }f_1\ f_2\texttt{)}]\!]s &= s[\mathbb{F}[\![f_1]\!] \mapsto v],\ v = s[\mathbb{F}[\![f_1]\!]] + s[\mathbb{F}[\![f_2]\!]] \\
\mathbb{C}[\![\texttt{(decrease }f_1\ f_2\texttt{)}]\!]s &= s[\mathbb{F}[\![f_1]\!] \mapsto v],\ v = s[\mathbb{F}[\![f_1]\!]] - s[\mathbb{F}[\![f_2]\!]] \\
\mathbb{C}[\![\texttt{(scale-up }f_1\ f_2\texttt{)}]\!]s &= s[\mathbb{F}[\![f_1]\!] \mapsto v],\ v = s[\mathbb{F}[\![f_1]\!]] * s[\mathbb{F}[\![f_2]\!]] \\
\mathbb{C}[\![\texttt{(scale-down }f_1\ f_2\texttt{)}]\!]s &= s[\mathbb{F}[\![f_1]\!] \mapsto v],\ v = s[\mathbb{F}[\![f_1]\!]] \ / \ s[\mathbb{F}[\![f_2]\!]] \\
\mathbb{C}[\![\texttt{(and }e_1\ ...\ e_n\texttt{)}]\!]s &= (\mathbb{C}[\![e_1]\!] \ ||\ ...\ ||\ \mathbb{C}[\![e_n]\!])s \\
\mathbb{C}[\![\texttt{(when }f\ e\texttt{)}]\!]s &= \mathbb{C}[\![e]\!]s \text{ if } s \models \mathbb{F}[\![f]\!], s \text{ otherwise} \\
\mathbb{C}[\![\texttt{(forall }v\ \texttt{-}\ t\ e\texttt{)}]\!]s &= (\mathbb{C}[\![e_1]\!] \ ||\ ...\ ||\ \mathbb{C}[\![e_n]\!])s \\
&\quad \text{where } e_1,...,e_n \in \{e[v/x] \mid \forall x : x \in \mathsf{objects}(s,t)\}
\end{aligned}
$$

Figure 3-6: Denotational semantics for effect expressions in PDDL, which map states $s \in \mathcal{S}$ to other states. The || symbol denotes parallel composition of effects.

universally-quantified effects using `forall` keyword, as introduced by Pednault's Action Description Language (ADL) [41]. These expressions allow for (infinitely) more concise specifications of actions (such as stating that *all* crates in a truck are unloaded in a certain city) albeit at the cost of additional work for automated planners. Finally, the numeric fragment of PDDL supports assignment and arithmetic modification of numeric fluents using keywords such as `assign` or `increase`.

PDDL.jl supports interpretation of all of these expressions, which it performs by walking the syntax tree and accumulating additions, deletions and assignments in a `Diff` data structure that represents the difference between the current and the subsequent states. Because PDDL actions are like atomic commands in programming, care is taken to ensure that effects represented by sub-expressions are composed in *parallel* with each other (denoted in Figure 3-6 using the || operator), as opposed to the standard sequential composition of commands in (non-concurrent) imperative programs. In particular, parallel composition is defined by performing all read operations (i.e. looking up fluent values) before any write operations (assignments, additions, deletions), which is achieved by accumulating all changes in the `Diff` structure and applying them at once. It is an error if any conflicting effects are composed (e.g. addition and deletion of the same fluent).

3.4 Extending Effects with Custom Imperatives

Certain planning domains require more general effects than can be supported by the evaluation of custom functions combined with the ability to assign state variables with the `assign` effect. For example, probabilistic effects in stochastic domains cannot be represented as pure deterministic functions without side effects. As such, the PDDL.jl interpreter is designed so that effects can be extended with custom imperatives as well. Using an extension interface similar to the one for custom functions, developers can easily write extension libraries that introduce new imperatives that the interpreter uses when constructing state `Diff`s from effect expressions.

Figure 3-7 shows an example of how the interpreter can be extended to support the `probabilistic` effect expression defined in Probabilistic PDDL (PPDDL) [12]. Given an expression of the form `(probabilistic` p_1 e_1 ... p_n e_n`)`, it samples the sub-expression e_i with probability p_i from categorical distribution and combines the effect of e_i with the `diff` computed so far. The PDDL interpreter is thus augmented with a sampling semantics that custom functions are not able to achieve, allowing it to simulate the transition dynamics of Markov Decision Processes.

```
"Sample a sub-effect from a categorical distribution."
function prob_effect!(diff::Diff, domain::Domain, state::State, effect::Term)
    total_prob = 0.0
    # Sample a random uniform variate u in [0, 1]
    u = rand()
    # Iterate over two arguments at a time
    for (prob, eff) in Iterators.partition(effect.args, 2)
        # Check if u lies in probability range
        if total_prob <= u < total_prob + prob
            # Combine sub-effect with current state difference
            combine!(diff, effect_diff(domain, state, eff))
            break
        end
        total_prob += prob
    end
    return diff
end

PDDL.register!(:effect, "probabilistic", prob_effect!)
```

Figure 3-7: The PDDL.jl interpreter can be extended to support custom effect expressions such as the `probabilistic` effect from PPDDL [12].

3.5 Interpreting Actions in Reverse

A final important feature of the built-in interpreter is that it can also interpret actions in *reverse* for the purely logical subset of PDDL domains, thereby implementing the `relevant` and `regress` interface methods for regression-based planners and heuristics. The interpreter does this by treating effect expressions as preconditions and precondition formulae as effects. If the effect of an action adds at least one predicate that is true in the current state (or alternatively, deletes at least one predicate that is currently false), deletes no predicates that are currently true, and adds nothing is currently false, then it is *relevant* to achieving that state. The PDDL interpreter implements this by checking whether the additions and deletions of an effect expression meet the requirement just stated.

To execute the action in reverse (i.e., to perform *regression*), we first consider the case where its precondition contains no disjunctions, i.e., it is a conjunction of either positive or negative literals. It must hence be the case that all of the positive literals hold true in the pre-image of the action, and that the negative literals hold false. We can thus reverse an action by setting literals in its precondition to true, and all negative literals to false. Apart from those literals, any predicates added or deleted by the action may be either false or true in its preimage, which is also called the *weakest precondition* of an action a relative to a state s. We can express this in set notation as follows:

$$\mathsf{preimage}(s, a) = \mathsf{facts}(s) \setminus \mathsf{add}(a) \cup \mathsf{precond}^{+}(a) \setminus \mathsf{precond}^{-}(a)$$

where $\mathsf{facts}(s)$ is the set of literals that hold true in s, $\mathsf{add}(a)$ the additions of a, and $\mathsf{precond}^{+}(a)$, $\mathsf{precond}^{-}(a)$ are the positive and negative preconditions respectively. Extending this to actions with disjunctive preconditions, we can normalize the precondition formula to disjunctive normal form (DNF), apply the same analysis as above, and take the union of the pre-images with respect to each precondition clause. The PDDL.jl interpreter implements this by returning a list of partially-specified states when `regress` is called, one for each disjunct.

Note that these implementations of action relevance and regression apply not just to concrete states, but also to abstract states specified by conjunctions of ground predicates, where each abstract state $\bar{s}$ corresponds to the *set* of concrete states where thase predicates hold true. As noted in Section 2.3.4, this is the primary setting of regression planners, which work backwards from a goal condition (equivalent to a set of goal states) by iteratively finding the weakest preconditions of actions (also specified by conjunctions of predicates) until a set of states is found which includes the initial state.

It is possible to further extend preimage computation for actions to more general settings such as numeric planning. However, this requires partial state representations capable of capturing more general constraints on the space of possible fluent values, for example, affine constraints on numeric variables. While PDDL.jl does not currently provide support for such state representations, it does provide the building blocks for reasoning about abstract sets of states in general. This functionality, known as abstract interpretation, is discussed by the next chapter in detail.

Chapter 4

Abstract Interpretation for Generalized Heuristic Search

Domain-general symbolic planners largely derive their generality by constructing effective search heuristics through formal analysis of symbolic world models, doing so without recourse to the domain-specific knowledge and learned expertise that humans often employ to solve problems. One approach to such heuristic construction is to consider planning in a *relaxed* or *abstracted* state space. By relaxing the problem, two major benefits are achieved: (i) the length of a solution to the relaxed problem can be used as an (optimistic) estimate of the distance to the goal, and thus provide heuristic search guidance; (ii) since the problem is relaxed, it can be much less costly to plan in the relaxed space relative to the original problem itself, ensuring that heuristic estimates can be computed sufficiently quickly to guide planning in the original problem. Indeed, this approach to heuristic computation has been so successful that it is now the dominant approach used by domain-general symbolic planners [15, 16, 17, 21].

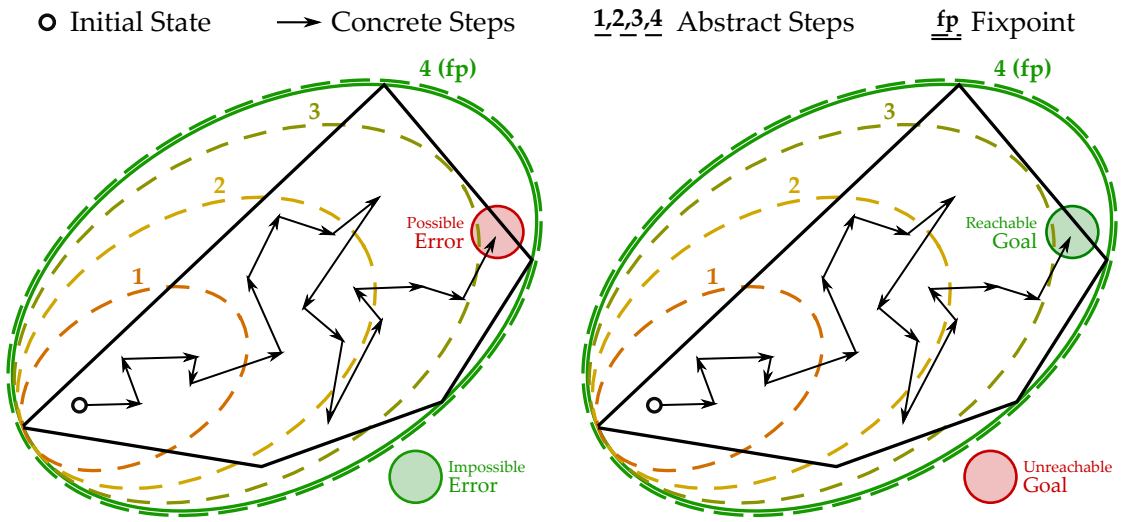
In deriving problem relaxations and abstractions for heuristic search, AI planning researchers have made connections to frameworks for abstraction in other fields, especially the field of model checking [55, 56]. A closely related framework for abstraction comes from the theory of abstract interpretation developed by Cousot and Cousot [57]. Abstract interpretation is a framework for the sound approximation of program semantics. Since symbolic actions can be understood as commands in a programming

language, plans are formally equivalent to programs. This suggests that ideas and innovations from abstract interpretation can be used in heuristics for planning as well.

In this chapter of the thesis, we develop the aforementioned line of thought in the context of PDDL, articulating the formal connection between abstract interpretation and problem abstraction for heuristic search. This connection is inspired by the semantics of domain abstraction developed in Planning Modulo Theories [33], and has been noted in other works on abstraction for planning [58]. However, the connection described here is, to the author's knowledge, the first to use the formalism of abstract interpretation proper, bringing with it many possibilities for re-envisioning and innovating upon existing approaches to domain abstraction. After making this connection, we describe how PDDL.jl implements these ideas by extending the interpreter described in Chapter 3 to serve as an abstract interpreter. This abstract interpreter automatically inherits the extensibility of the generic interpreter through custom functions. In addition, users can provide custom abstractions, such as the interval abstraction for numeric fluents. This allows heuristic computation to be extended to new data types, further increasing the generality of domain-general planning. Finally, we discuss additional ways that abstract interpretation might benefit automated planning, and how PDDL.jl enables research into these possibilities.

4.1 Abstract Semantics for Symbolic Planning

Abstract interpretation works by over-approximating the set of possible ways a program could branch and evolve as it is executed, perhaps because the program's arguments are unknown, or because there are non-deterministic choice points in the program (corresponding to e.g., input from a user) [57, 59]. This is useful for error detection and verifying correctness. If we can over-approximate all of the ways a program can run, and check that this never results in an error, then the original (unapproximated) program must never reach an error. Otherwise, an error is possible, and a warning can be raised to the programmer. Figure 4-1(a) depicts this graphically, with arrows ($\rightarrow$) corresponding to steps along one possible execution path,



(a) Abstract Interpretation for Error Detection and Formal Verification

(b) Abstract Planning for Goal Reachability and Heuristic Estimation

Figure 4-1: A graphical analogy between (a) abstract interpretation for error detection and (b) abstract planning for determining the reachability of a goal. In abstract interpretation, over-approximating the set of reachable states via abstract steps allows us to rule out impossible errors. In abstract planning, we can similarly rule out some goals as unreachable, or otherwise estimate how many steps it takes to reach them.

and the solid lines bounding all possible ways the program could execute. Abstract interpretation over-approximates this set through a series of *abstract steps* (dashed ovals): whereas a concrete step moves the program's state from one specific point to another, an abstract step updates the *set* of states in which the program could be. This set of states typically grows with more steps, until a fixpoint (solid oval) is reached, at which point further steps make no difference. Having computed this fixpoint, we are guaranteed that any states outside of it are unreachable, including any errors they contain. However, states (and errors) that lie within the fixpoint may still be possible, depending on whether they lie within the true set of reachable states.

As Figure 4-1(b) shows, this process of abstract interpretation is applicable to symbolic planning with one simple change: instead of trying to detect errors, we are trying to reach goals. Concrete steps now correspond to symbolic actions, and program branch points to choices over such actions. If we run this process and find that the goal set lies outside the fixpoint, we can conclude that it is unreachable and give up on planning. More interestingly, we can use the results of this abstract

```julia
function nd_search(domain, state, goal)
    # Loop until goal is reached
    while !satisfy(domain, state, goal)
        # Choose and execute an available action
        act = choice(available(domain, state)...)
        state = execute(domain, state, act)
    end
end
```

(a) Julia implementation

```lisp
(defun nd-search (goal)
  ;; Loop until goal is reached
  (loop until goal do
    ;; Get list of available actions
    (let (actions (list-available))
      ;; Choose and execute an action
      (apply choice actions))))
```

(b) Lisp-style implementation

Figure 4-2: State space planning as non-deterministic search. We show (a) a Julia implementation using the PDDL.jl interface and (b) a Lisp-style implementation inspired by ALisp [60]. The `choice` function indicates non-deterministic choice, and `apply` applies this function to a list of arguments.

analysis even if we do not rule out a goal. Since abstract planning is approximate and hence less difficult than planning in the original space, the number of abstract steps to a goal (three, in the context of Figure 4-1) serves as an optimistic estimate of the true number of (concrete) steps required to reach the goal. As such, we can use this number as heuristic guidance within a symbolic planning algorithm.

To formalize this intuitive picture, we need to clarify how the process of planning can be viewed as executing a program. Figure 4-2 shows how planning can be framed as a non-deterministic search procedure, which iteratively explores all possible paths through state space until a goal predicate (`goal`) is satisfied. A program path that successfully reaches the goal constitutes a plan, and the length of that path is the length of the plan. On the left, we show a Julia implementation using the PDDL.jl interface. On the right, we show an equivalent Lisp-style implementation inspired by the ALisp language for programmable reinforcement learning [60], which introduces the `(choice` a_1 `...` a_n`)` expression for non-deterministic execution of one of the actions a_1 to a_n. We also introduce the `(list-available)` function, which returns all available actions in the current state. We provide this implementation for consistency with the Lisp-style syntax of PDDL predicates and actions. This allows us to conveniently interpret PDDL predicates like `(on a b)` as Boolean expressions in this ALisp-like language, and (ground) actions like `(stack a b)` as imperative commands that modify the state. Note that we omit the domain and initial state in this Lisp-style implementation, which are implicitly assumed.

By framing state space search as a non-deterministic program, we can now use abstract interpretation to ask various questions about the program, and hence about search. For example:

- Will the search loop ever terminate under any choice of branches? (Is the goal predicate ever reachable, under any choice of actions?)
- If the loop terminates, can we bound the minimal number of iterations? (What is a lower bound on the length of the shortest plan to the goal?)

Indeed, we can use abstract interpretation to answer questions about many other programs besides the procedure in Figure 4-2. For example, we could express programs that correspond to generalized plans or policies that use control flow to solve multiple problem instances [60, 61, 62], then use abstract interpretation to check for correctness. Abstract interpretation could even help to synthesize such programs [63, 64, 65]. However, we restrict our attention for now to the program in Figure 4-2, because our primary focus is on deriving heuristics to speed up search for specific plans.

Having fixed our program of interest, we now need to specify (i) what it means to execute multiple branches of this program; and (ii) how to efficiently over-approximate this set of possible executions, since this is intractable to compute directly due to exponential growth. To address (i) we need a *collecting semantics* that defines what happens when multiple branches (i.e. actions) are simultaneously executed. To address (ii) requires specifying an *abstract semantics* that over-approximates the collecting semantics. We provide these semantics in Figure 4-3. For the collecting semantics (Figure 4-3(a)), we first define the concrete semantic function $\mathbb{E}[\![\cdot]\!]$ for how expressions like goal formulae, built-in arithmetic functions, or `list-available` are evaluated in a state $s \in S$. Next, we lift the effect semantic function $\mathbb{C}[\![\cdot]\!]$ defined in Figure 3-6 to operate over *sets* of states $S \in \mathcal{P}(\mathcal{S})$, since we want to keep track of the states along multiple branches. We then define non-deterministic execution of some action (which we treat as synonymous with its effects) via `choice`: Since any choice is possible, we take the union of all possible resulting outputs. We handle branching at the head of a `loop` similarly, per the least fixpoint semantics given in [59].

$\underline{\mathbb{E}[\![expr]\!] : \mathcal{S} \to \mathcal{V}}$

$\mathbb{E}[\![f]\!]s$	$=$	$s \models \mathbb{F}[\![f]\!]$ where f is a logical formula
$\mathbb{E}[\![(B\ x_1\ x_2)]\!]s$	$=$	standard interpretation, where $B \in \{+, -, *, /, =, ...\}$
$\mathbb{E}[\![(F\ a_1\ ...\ a_n)]\!]s$	$=$	user-defined, where F is a custom function
$\mathbb{E}[\![\texttt{(list-available)}]\!]s$	$=$	$\{a \in \mathcal{A} \mid \mathbb{E}[\![\mathsf{precond}(a)]\!]s\}$

$\underline{\mathbb{C}[\![expr]\!] : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})}$

$\mathbb{C}[\![e]\!]S$	$=$	$\cup\{\mathbb{C}[\![e]\!]s \mid s \in S\}$ where e is an effect expression
$\mathbb{C}[\![a]\!]S$	$=$	$\mathbb{C}[\![\mathsf{effect}(a)]\!]S$ where a is a ground action
$\mathbb{C}[\![\texttt{(choice}\ a_1\ ...\ a_n\texttt{)}]\!]S$	$=$	$\mathbb{C}[\![a_1]\!]S \cup\ ...\ \cup \mathbb{C}[\![a_n]\!]S$
$\mathbb{C}[\![\texttt{(loop until}\ f\ \texttt{do}\ c\texttt{)}]\!]S$	$=$	$\{s \in$ least fixpoint of $T \mid \mathbb{E}[\![f]\!]s\}$
		where $T(X) = S \cup \mathbb{C}[\![c]\!]\{s \in X \mid \neg\mathbb{E}[\![f]\!]s\}$

(a) Concrete collecting semantics for expressions $\mathbb{E}[\![\cdot]\!]$ and commands $\mathbb{C}[\![\cdot]\!]$.

$\underline{\mathbb{E}^\sharp[\![expr]\!] : \mathcal{X}^\sharp \to \mathcal{U}^\sharp}$

$\mathbb{E}^\sharp[\![f]\!]S^\sharp$	$=$	$\cup\{s \models \mathbb{F}[\![f]\!] \mid s \in \gamma(S^\sharp)\}$ where f is a logical formula
$\mathbb{E}^\sharp[\![(B\ x_1\ x_2)]\!]S^\sharp$	$=$	user-defined, where $B \in \{+, -, *, /, =, ...\}$
$\mathbb{E}^\sharp[\![(F\ a_1\ ...\ a_n)]\!]S^\sharp$	$=$	user-defined, where F is a custom function
$\mathbb{E}^\sharp[\![\texttt{(list-available)}]\!]S^\sharp$	$=$	$\{a \in \mathcal{A} \mid \texttt{true} \in \mathbb{E}^\sharp[\![\mathsf{precond}(a)]\!]S^\sharp\}$

$\underline{\mathbb{C}^\sharp[\![expr]\!] : \mathcal{X}^\sharp \to \mathcal{X}^\sharp}$

$\mathbb{C}^\sharp[\![\texttt{(}P\ a_1\ ...\ a_n\texttt{)}]\!]S^\sharp$	$=$	$S^\sharp[P(a_1, ..., a_n) \mapsto \{\texttt{true}\}]$
$\mathbb{C}^\sharp[\![\texttt{(not (}P\ a_1\ ...\ a_n\texttt{))}]\!]S^\sharp$	$=$	$S^\sharp[P(a_1, ..., a_n) \mapsto \{\texttt{false}\}]$
$\mathbb{C}^\sharp[\![\texttt{(assign}\ f_1\ f_2\texttt{)}]\!]S^\sharp$	$=$	$S^\sharp[\mathbb{F}[\![f_1]\!] \mapsto v],\ v = \mathbb{E}^\sharp[\![f_2]\!]$
$\mathbb{C}^\sharp[\![\texttt{(and}\ e_1\ ...\ e_n\texttt{)}]\!]S^\sharp$	$=$	$(\mathbb{C}^\sharp[\![e_1]\!]\|\|...\|\|\mathbb{C}^\sharp[\![e_n]\!])S^\sharp$
$\mathbb{C}^\sharp[\![a]\!]S^\sharp$	$=$	$\mathbb{C}^\sharp[\![\mathsf{effect}(a)]\!]S^\sharp$ where a is a ground action
$\mathbb{C}^\sharp[\![\texttt{(choice}\ a_1\ ...\ a_n\texttt{)}]\!]S^\sharp$	$=$	$\mathbb{C}^\sharp[\![e_1]\!]S^\sharp \cup^\sharp\ ...\ \cup^\sharp \mathbb{C}^\sharp[\![e_n]\!]S^\sharp$
$\mathbb{C}[\![\texttt{(loop until}\ f\ \texttt{do}\ c\texttt{)}]\!]S^\sharp$	$=$	$\lim_{n\to\infty}(T^\sharp)^n(S^\sharp)$ where $T^\sharp(X^\sharp) = S^\sharp \cup^\sharp \mathbb{C}^\sharp[\![c]\!]X^\sharp$

(b) Non-relational abstract semantics for expressions $\mathbb{E}^\sharp[\![\cdot]\!]$ and commands $\mathbb{C}^\sharp[\![\cdot]\!]$.

Figure 4-3: Concrete collecting semantics and abstract semantics for PDDL with `list-available`, `choice` and `loop` expressions. We adopt a non-relational (a.k.a Cartesian) abstraction for the abstract semantics, where an abstract state $S^\sharp$ is simply a tuple of abstract values for each fluent. Abstractions for each value type (e.g. integers, sets, etc.) and corresponding functions are user defined.

We next define generic abstract semantics which over-approximate the collecting semantics, provided in Figure 4-3(b). These semantics operate over abstract states $S^\sharp \in \mathcal{X}^\sharp$. Each abstract state $S^\sharp$ is intended to over-approximate a set of concrete states $S \in \mathcal{P}(\mathcal{S})$ across all branches at a particular stage of program execution. To be precise, $S^\sharp$ is a *sound* over-approximation of S if the set of concrete states it represents, $\gamma(S^\sharp)$, is a superset of S, i.e., $S \subseteq \gamma(S^\sharp)$. The abstract semantics for expressions are defined by $\mathbb{E}^\sharp[\![\cdot]\!]$, specifying how an expression maps an abstract state $S^\sharp \in \mathcal{X}^\sharp$ to an abstract value $V^\sharp \in \mathcal{U}^\sharp$, while the abstract semantics for commands are defined by $\mathbb{C}^\sharp[\![\cdot]\!]$, specifying how a command maps one abstract state to another. Of note is the abstraction for `(choice ...)` over actions, which takes the abstract union $\cup^\sharp$ over the results of all actions. Similarly, the `(loop ...)` abstraction takes the repeated abstract union of $S^\sharp$ pre- and post-iteration. This differs from the collecting semantics because the abstract union $\cup^\sharp$ over-approximates the ordinary union $\cup$, making it easier to compute than tracking an explicit union of values across program branches. How this is exactly achieved depends on the choice of abstraction, which we discuss next.

As shown in Figure 4-3(b), some aspects of the abstract semantics for PDDL are user-defined. Because there are many ways to over-approximate states, the choice of abstraction can be customized. The only major choice that we make is adopting abstract semantics that are *non-relational* or *Cartesian* [56]: Each abstract state $S^\sharp$ is just a tuple $(f_1 \mapsto V_1^\sharp, ..., f_k \mapsto V_k^\sharp)$ of the abstract values $V_i^\sharp$ assigned to each fluent f_i that the state is defined over. As such, the set of concrete states $\gamma(S^\sharp)$ represented by $S^\sharp$ is the Cartesian product of sets of concrete values $\gamma(V_i^\sharp)$ represented by each abstract value $V_i^\sharp$. Furthermore, the abstract union $\cup^\sharp$ of states is just the tuple of abstract unions of their values. We make this choice for ease of implementation, since it means that once we specify sound abstractions for fluent values, we automatically have a sound abstraction over states. For example, we could abstract over Booleans with their powerset (Figure 4-4(a)), allowing predicates to be both false and true [66, 33], and over numeric fluents using the interval abstraction (Figure 4-4(b)), which over-approximates a set A of numbers by the interval $[\inf A, \sup A]$ [33, 21]. Both of these are sound by construction, and hence their Cartesian product is as well.

$B \in \mathcal{P}(\mathbb{B})$	$= \{\mathsf{none}, \mathsf{false}, \mathsf{true}, \mathsf{both}\}$		
$\gamma(B)$	$= B$	$\alpha(B)$	$= B$
$B \sqsubseteq \mathsf{both}$	$\iff \texttt{true}$	$\mathsf{none} \sqsubseteq B$	$\iff \texttt{true}$
$\mathsf{false} \sqcup \mathsf{true}$	$= \mathsf{both}$	$B \sqcup B$	$= B$
$\mathsf{both} \sqcup \mathsf{B}$	$= \mathsf{both}$	$\mathsf{none} \sqcup \mathsf{B}$	$= B$
$\mathsf{both} \wedge B$	$= (\mathsf{false} \wedge B) \sqcup (\mathsf{true} \wedge B)$	$\mathsf{none} \wedge B$	$= \mathsf{none}$
$\mathsf{both} \vee B$	$= (\mathsf{false} \vee B) \sqcup (\mathsf{true} \vee B)$	$\mathsf{none} \vee B$	$= \mathsf{none}$
$\neg\ \mathsf{both}$	$= \mathsf{both}$	$\neg\ \mathsf{none}$	$= \mathsf{none}$

(a) Powerset abstraction for Boolean values $\mathbb{B}$ (a.k.a. four-valued logic)

$I \in \mathcal{I}$	$= \{[a,b] \mid a,b \in \mathbb{R} \cup \{-\infty, +\infty\}, a \leq b\}$		
$\gamma([a,b])$	$= \{x \in \mathbb{R} \mid a \leq x \leq b\}$	$\alpha(A)$	$= [\inf A, \sup A],\ A \subseteq \mathbb{R}$
$[a,b] \sqcup [c,d]$	$= [\min(a,c), \max(b,d)]$	$[a,b] \sqsubseteq [c,d]$	$\iff (a \geq c) \wedge (b \leq d)$
$[a,b] + [c,d]$	$= [a+c, b+d]$	$[a,b] - [c,d]$	$= [a-d, b-c]$

(b) Interval abstraction for real-valued fluents (with + and − operations).

Figure 4-4: Commonly used abstractions of fluent values. For each abstraction, we provide the concretization γ and abstraction α functions, the partial order $\subseteq$, and the least-upper-bound $\sqcup$, which plays the role of the abstract union $\cup^\sharp$.

How should a user decide between different value abstractions? In general, one should consider the soundness of the approximation, its precision, and its efficiency. Soundness can be achieved by ensuring that all abstractions g of concrete operations f maintain the over-approximation, i.e. $\forall S^\sharp \in \mathcal{X}^\sharp : f(\gamma(S^\sharp)) \subseteq \gamma(g(S^\sharp))$. This is the case when using the value abstractions in Figure 4-4 with the Cartesian abstraction in Figure 4-3 (we omit proof since these choices are standard, see [59]). The tradeoff between precision and efficiency is harder to navigate: precision means that abstract computation (including heuristic estimation) is more accurate, but at the expense of slower computation. For example, the powerset abstraction is exact, but becomes intractable as the cardinality of the underlying set increases. In contrast, the abstraction which includes all possible values is useless, but trivial to compute. Good choices like the interval abstraction lie somewhere in between, and can even be automatically constructed. We refer the reader to [33] and [56] for some possibilities.

4.2 Deriving Heuristics through Abstraction

With our abstract semantics defined, we can revisit how this allows us to determine goal reachability or heuristic distance. Returning to the `nd-search` procedure in Figure 4-2(b), we can imagine first abstracting the initial state $S_0^\sharp = \alpha(\{s_0\})$, where α is an abstraction function (that maps e.g. numbers to intervals). We then repeatedly apply the abstract semantics for `(list-available)` and `(choice ...)`, the composition of which we denote as the abstract transition $T^\sharp$. With each iteration i, the abstract state $S_i^\sharp = T^\sharp(S_{i-1}^\sharp)$ grows, because we keep merging the results of executing all available actions passed to `(choice ...)`, which leads to more preconditions being satisfied, and hence to more available actions being returned by `(list-available)`. This eventually leads to one of the following outcomes:

1. A fixpoint $F^\sharp = T(F^\sharp)$ is reached without satisfying the goal, in which case the goal can be ruled out as unreachable from s_0.

2. The abstract state $S_i^\sharp$ grows until the goal is satisfied, and the search procedure terminates (because the loop condition is met).

3. Neither of the above occur. The abstract state keeps growing without reaching a fixpoint, and abstract execution does not terminate.

In the first case, we can use the fact that the goal is unreachable from s_0 to avoid exploring that state during a concrete search procedure, assigning it a heuristic distance of ∞. In the second case, we can keep track of the number of iterations it takes for the goal to be reached, and use that number as a heuristic distance estimate to the goal. Indeed, as shown in [33], the heuristic returned by this procedure is exactly equivalent to the $h_{\max}$ and h_1 heuristics [67] when the powerset abstraction is used for Boolean values, and also equivalent to numeric extensions of $h_{\max}$ when the interval abstraction is used [21]. We call this the *reachability heuristic*, denoted h^A_{reach}, which is parameterized by an abstraction A. The power of the reachability heuristic lies in the fact that it is extremely general and customizable: Once a user provides sound abstractions A for the built-in functions in PDDL (arithmetic, etc.) and any

Algorithm 3 Reachability heuristic h^A_{reach} implemented using PDDL.jl.

```
function h_reach(domain::Domain, state::State, goal::Term, abstractions)
    # Construct abstract domain and abstract state
    absdom, abstate = PDDL.abstracted(domain, state; abstractions)
    # Iterate until we reach the goal or a fixpoint
    steps = 0
    while !(PDDL.satisfy(absdom, abstate, goal))
        steps += 1
        # Iterate over available actions
        next_abstate = abstate
        for act in PDDL.available(domain, state)
            # Compute least upper bound with each successor state
            next_abstate = PDDL.lub(next_abstate,
                                    PDDL.transition(domain, abstate, act))
        end
        # Terminate if fixpoint is reached
        if next_abstate == abstate return Inf end
        abstate = next_abstate
    end
    return steps
end
```

custom functions and types they wish to support (sets, arrays, etc.), h^A_{reach} can be used as a search heuristic over the corresponding concrete domain. This basic idea is due to Planning Modulo Theories [33], but they do not use abstract interpretation as a framework for customizing abstractions and checking their soundness. In contrast, PDDL.jl enables an implementation of h^A_{reach} that uses abstract interpretation, which we list in Algorithm 3. As can be seen, this implementation closely follows the abstract execution of `nd-search`. We first construct an abstract domain and state with `abstractions` that users can customize, then apply the same PDDL.jl interface methods as used for concrete domains. To merge the results of available actions, we use the `lub` function, which implements the abstract union $\cup^\sharp$ (a.k.a. the least upper bound). At each iteration, we check if the goal is reached, and return the number of steps if so. Otherwise, we return ∞ if a fixpoint is reached.

One subtle issue with Algorithm 3 is that, as written, it ignores the possibility of the third outcome mentioned above, where neither a goal nor a fixpoint is reached. This can occur when one of the value abstractions exhibits *infinite chains*: ordered sequences of abstract values $V_1^\sharp \sqsubseteq V_2^\sharp \sqsubseteq V_3^\sharp \ldots$ which have infinite length. For example,

if there is a numeric fluent f with an initial abstract value of $[2, 2]$, and an action a whose concrete semantics increase the value of f by 2, then abstract execution after i iterations will lead to abstract value of $[2, 2 + 2 * i]$, which will never converge to a fixpoint. If the goal additionally requires that $f = 0$, then the abstract analysis will never reach this goal, and hence fail to terminate.

To address this issue, most abstract semantics introduce a *widening* operator $\nabla : \mathcal{X}^\sharp \times \mathcal{X}^\sharp \to \mathcal{X}^\sharp$ that is applied at the head of loop constructs [57, 59], so called because it widens the set of represented values at its point of application. The only conditions for a widening operator are that:

1. It upper bounds its inputs: $X^\sharp, Y^\sharp \sqsubseteq X^\sharp \nabla Y^\sharp$
2. Repeated application leads to convergence: for any sequence $Y_1^\sharp, Y_2^\sharp, ... \in \mathcal{X}^\sharp$, computing $X_{i+1}^\sharp := X_i^\sharp \nabla Y_i^\sharp$ leads to a fixpoint after finite iterations.

As an example, a naive widening operator for the interval abstraction $[a, b] \nabla [c, d]$ would be to return $[a, b]$ if $[c, d] \sqsubseteq [a, b]$, and $[-\infty, \infty]$ otherwise. This is highly imprecise as an approximation, but ensures convergence. Whatever our choice of ∇, applying it at the start of each loop iteration resolves the non-termination issue. This means updating the semantics for `(loop until` f `do` c`)` in Figure 4-3 as follows:

$$\begin{aligned} \mathbb{C}[\![\texttt{(loop until } f \texttt{ do } c\texttt{)}]\!] S^\sharp &= \lim_{n \to \infty} (T^\sharp)^n (S^\sharp) \\ &\quad \text{where } T^\sharp(X^\sharp) = X^\sharp \nabla (S^\sharp \cup^\sharp \mathbb{C}^\sharp[\![c]\!] X^\sharp) \end{aligned}$$

By including the widening operator in the loop transition $T^\sharp$, we guarantee that repeated application of T converges in finite iterations. PDDL.jl also includes the `widen(absdom::Domain, s, s2)` method, which applies a custom widening associated with the abstract domain `absdom` to states `s1` and `s2`. Adding `widen` before the fixpoint check in Algorithm 3 thus addresses the issue in code.

With widening operators in hand, we can ask the question of what widening to choose so that h^A_{reach} is both efficient and informative. If ∇ is too aggressive, as in the naive example given earlier, then the loop could terminate in as little as one iteration,

leading to an admissible but highly uninformed heuristic that severely underestimates the true distance to the goal. If ∇ is too gradual, however, then the loop may converge too slowly, leading to inefficiency. To navigate this trade-off, we can draw inspiration from widening techniques used in the abstract interpretation literature [59]:

- **Delayed widening.** Instead of widening immediately, compute the least-upper-bound for the first n iterations, and apply widening only afterwards.
- **Widening with thresholds.** For interval abstractions, we can first widen $[a, b]\nabla[c, d]$ to $[-T_1, T_1]$ if $[c, d] \not\sqsubseteq [a, b]$, then to a wider interval $[-T_2, T_2]$ if this happens again, and so on until we finally widen to $[-\infty, \infty]$. The thresholds $T_1, ..., T_N$ can be user-specified, or come from an exponential ramp $2^1, ..., 2^N$.

Delayed widening is equivalent to terminating the loop in Algorithm 3 after a fixed number of iterations, and hence a natural choice that avoids over-relaxing the problem while limiting the amount of computation. To the author's knowledge however, equivalent methods to widening with thresholds have not been explored in the planning literature (besides the single-threshold case mentioned in [33]). This suggests that there are potential gains to be made by using this technique within heuristic computation, among many other innovations in abstract interpretation.

We close this section by remarking that, while we have focused on the reachability heuristic as presented in Algorithm 3, there are many other heuristic algorithms that can be reframed in light of abstract interpretation. For one, while the $h_{\max}$ algorithm is mathematically equivalent to h^A_{reach} under the Boolean powerset abstraction, implementations of the former usually perform Djikstra search over the relaxed planning graph of actions and preconditions, leading to greater efficiency. As shown by [19] and [21], this graph search procedure can be modified to accommodate interval abstractions – and by using the framework introduced here, any custom data type and corresponding abstraction. This gives not only generalized versions of $h_{\max}$, but also generalized versions of the additive heuristic h_{add}, which is typically much more informative [15]. Abstract interpretation may also prove useful for generalizing heuristics based on counter-example guided abstraction refinement [56] to problems

with continuous values or other data types. Finally, we have thus far only considered non-relational state abstractions for heuristic derivation. Relational abstractions (e.g. polyhedra [68]) may allow non-Cartesian abstractions such as the merge-and-shrink heuristic [69] to be extended beyond propositional domains. By illustrating the formal relationship between abstract interpretation and abstraction-based heuristics, we hope that many more of these fruitful connections can be explored in the future.

4.3 Abstraction via Function Overloading

Having shown how abstract interpretation can be used to derive and implement abstraction-based heuristics, we now turn to how PDDL.jl implements abstract interpretation itself. We adopt an elegant approach that exploits Julia's support for multiple dispatch, enables user customization, and operates with minimal modification to the concrete interpreter (Chapter 3): Introducing Julia data types to represent abstract values (e.g. real-valued intervals), and overloading functions that are defined over concrete values so that they implement the abstract semantics when given abstract values as arguments.

As an example of this general approach, Figure 4-5 shows the implementation of the interval abstraction in PDDL.jl. We use the `IntervalAbs` data type to represent abstract interval values, and implement the least upper bound `lub`, greater lower bound `glb`, and arithmetic operations $+, -, \times, /$ by forwarding method calls to the `Interval` type from IntervalArithmetic.jl [70]. Since the PDDL.jl interpreter evaluates arithmetic expressions by calling their corresponding Julia functions (`Base.:+(a, b)` for `(+ a b)`, etc.,) this means that if the interpreter encounters a numeric expression containing fluents with values of type `IntervalAbs`, it will automatically dispatch to the arithmetic methods defined over `IntervalAbs` arguments, and perform the necessary interval arithmetic. The primary advantage of this implementation strategy is that it is highly customizable and extensible. Suppose we want to use a different abstraction over the real numbers, such as the sign abstraction. To implement this abstraction, we just need to introduce a new Julia type, `SignAbs`, and implement the

```
struct IntervalAbs{T <: Real}
    interval::IntervalArithmetic.Interval{T}
end
IntervalAbs(a::T, b::T) where {T <: Real} =
    IntervalAbs{T}(IntervalArithmetic.Interval{T}(a, b))
IntervalAbs(x::T) where {T <: Real} =
    IntervalAbs{T}(x, x)
PDDL.lub(a::IntervalAbs, b::IntervalAbs) =
    IntervalAbs(union(a.interval, b.interval))
PDDL.glb(a::IntervalAbs, b::IntervalAbs) =
    IntervalAbs(intersect(a.interval, b.interval))
Base.:-(a::IntervalAbs) = IntervalAbs(-a)
Base.:+(a::IntervalAbs, b::IntervalAbs) = IntervalAbs(a.interval + b.interval)
Base.:-(a::IntervalAbs, b::IntervalAbs) = IntervalAbs(a.interval - b.interval)
Base.:*(a::IntervalAbs, b::IntervalAbs) = IntervalAbs(a.interval * b.interval)
Base.:/(a::IntervalAbs, b::IntervalAbs) = IntervalAbs(a.interval / b.interval)
```

Figure 4-5: PDDL.jl implementation of the interval abstraction. We introduce a new type `IntervalAbs`, and implement its semantics by forwarding method calls to the `Interval` type from IntervalArithmetic.jl.

.

same arithmetic and ordering functions shown in Figure 4-5. Alternatively, suppose that we want to define an abstraction over the custom set theory introduced in Figure 3-4. Then we can just define an abstraction type, `SetAbs`, and implement all the functions that constitute the theory (`cardinality`, `member`, `union`, `intersect`, etc.) for this new type.

Given these value abstractions, we can construct abstract domains and states using the `abstracted` function shown at the top of Algorithm 3. The `abstracted(domain, state; abstractions)` function accepts a concrete `domain` and `state` as input, as well as a dictionary of `abstractions` that map type symbols to abstraction types (e.g. `:boolean => BooleanAbs`, `:numeric => IntervalAbs`). By changing the entries of this dictionary, users can customize which abstractions are used for each concrete value type. This means that, unlike many other planning systems that support some form of abstraction, PDDL.jl users are not beholden to any particular value abstraction, and are free to customize and experiment with different abstractions to improve performance on downstream tasks. In the future, we aim to support even more customization, such as allowing abstractions to be specified on a per-fluent basis, allowing PDDL.jl to serve as a platform for even more uses of abstract interpretation.

4.4 Other Applications of Abstract Interpretation

In this chapter, we have focused on how abstract interpretation can be used to derive more general planning heuristics for forward state space search. However, there are many other possible applications of abstract interpretation that could be enabled by PDDL.jl, which we briefly reflect upon here.

4.4.1 Constructing Hierarchies of Abstractions

One of the earliest uses of abstraction in planning was to automatically construct hierarchies of abstract domains, allowing planners to solve problems in a coarse-to-fine manner by first planning at the most abstract level, and then refining plans at more and more concrete levels [71, 72]. Similar ideas have much more recently been explored in counter-example guided abstraction refinement for classical planning [56], which is inspired by the model-checking literature. Given these connections, it is possible that the process of hierarchical domain abstraction could be reframed using the theory of abstract interpretation, allowing for new insights about how to automatically derive the best abstraction hierarchies for planning.

4.4.2 Reverse Abstract Interpretation for Bidirectional Search

As noted briefly at the end of Chapter 3, abstract interpretation can also be executed in reverse [73, 74], allowing for generalizations of the pre-image semantics described in Section 3.5 to numeric and other domains, as well as efficient over-approximations of these inverse semantics. While PDDL.jl currently does not support abstract interpretation in reverse, implementing this functionality might allow for the development of new regression search algorithms that operate over more general domains, as well as novel combinations of (abstract) backwards search with (concrete) forward search. Such combinations have been explored in both task-and-motion planning [48] and program analysis [75], suggesting that there is room for an interdisciplinary exchange of ideas, and perhaps the development of a framework that allows for the systematic exploration of these bi-directional search algorithms.

4.4.3 Abstract Interpretation for Generalized Planning

Finally, as mentioned earlier in this chapter, abstract interpretation could be used in the verification and synthesis of generalized plans, which often take the form of imperative programs with control flow [60, 61, 62]. Indeed, with minimal modifications, the abstract semantics presented in Figure 4-3 could be used for the verification of ALisp-like policy programs similar to the `nd-search` procedure in Figure 4-2(b). Support for abstraction-guided program synthesis, on the other hand, will likely require more specific infrastructure to be built. Nonetheless, given the many applications of abstract interpretation to program synthesis outside of symbolic planning [63, 64, 65], we anticipate that many of the ideas and connections we have developed here will prove useful in fostering the convergence of these closely related fields.

Chapter 5

Domain Compilation and Static Analysis for Efficient Planning

High performance systems for symbolic planning typically reduce computational cost in multiple ways. In Chapter 4, we described one such approach that PDDL.jl enables: Deriving heuristics that make search smarter, and hence reduce the number of search operations necessary. In this chapter, we describe two other approaches supported by PDDL.jl: Domain compilation to reduce the runtime and memory footprint of each basic search operation, as well as static analysis to prune irrelevant domain information and hence reduce the size of the search space.

Compilation is implemented by generating domain-specific state representations and method definitions for each action in the domain. Since the targets of compilation are concrete implementations of the abstract data types and methods that comprise the PDDL.jl interface, users and downstream code can make use of compiled code as a drop-in replacement for the generic PDDL interpreter (see Figure 5-1), achieving constant factor speed-ups of an order-of-magnitude or more. Static analysis tools are provided for downstream use (e.g. ignoring irrelevant actions in planning heuristics), and can also be used as part of the compilation process. The following sections introduce the two main compilation techniques provided by PDDL.jl's built-in compiler, followed by a description of static analysis tools and other compilation techniques, and finally a discussion of how the semantics of the compiler can be extended.

```julia
# Load a generic representation of PDDL domain and problem
domain = load_domain("blocksworld.pddl")
problem = load_problem("blocksworld-problem.pddl")

# Compile the domain and problem to get a compiled domain and state
c_domain, c_state = compiled(domain, problem)

# Perform breadth-first search on the compiled domain using Algorithm 1
plan = bfs(c_domain, problem)

# Execute the plan on the compiled initial state
for act in plan
    c_state = transition(c_domain, c_state, act)
end

# Check that the goal is achieved in the final state
@assert satisfy(c_domain, c_state, get_goal(problem))
```

Figure 5-1: Using PDDL.jl's built-in compiler via the `compiled` method. The compiled domain `c_domain` can be used in place of the generic `domain` representation loaded from a PDDL file in a planning algorithm such as Algorithm 1

.

5.1 Compiled State Representations

The symbolic states manipulated by PDDL.jl (Definition 2.1.2) are composed of ground fluents (represented as `Term`s in PDDL.jl) and their corresponding valuations. As such, they can be generically implemented using hash tables, using Julia's `Set` data type for Boolean fluents that are true, and the `Dict` data type for all other fluents (Figure 5-2(a)). This is the representation used by the PDDL.jl interpreter. However, while this implementation is straightforward, it is not very efficient. Accessing the value of a fluent requires hashing a `Term`, which accumulates with many accesses, and the memory overhead of maintaining a hash table can be quite significant.

To avoid these overheads the PDDL.jl compiler generates *compiled state representations* that are specialized to a particular domain and problem. This representation takes advantage of the fixed number of objects in standard PDDL problems, allowing for the generation of finite-object state representations with a known size in advance, where Boolean fluents can be represented as bit arrays over their object domains, and other fluents can be represented as standard arrays of the appropriate type.

```
GenericState
    types -> Set{Compound} with 3 elements
       pddl"(block a)",
       pddl"(block b)",
       pddl"(block c)"
    facts -> Set{Term} with 7 elements
       pddl"(handempty)",
       pddl"(clear a)",
       pddl"(clear b)",
       pddl"(clear c)"
       pddl"(ontable a)",
       pddl"(ontable b)",
       pddl"(ontable c)"
    values -> Dict{Term,Any} with 0 entries

Size: 1720 bytes
Median Access Time: 394 ns
```

(a) Generic state representation

```
CompiledBlocksworldState
    handempty ->
        true
    clear -> 3-element BitVector
        1  1  1
    holding -> 3-element BitVector
        0  0  0
    ontable -> 3-element BitVector
        1  1  1
    on -> 3x3 BitMatrix
        0  0  0
        0  0  0
        0  0  0

Size: 336 bytes
Median Access Time: 58.5 ns
```

(b) Compiled state representation

Figure 5-2: Generic vs. compiled state representations in the Blocksworld domain.

An example is shown in Figure 5-2(b) for the Blocksworld domain in a problem with three blocks. Rather than storing each true predicate in a `Set`, as in the generic state representation, the compiled representation stores 0-place predicates (e.g. `(handempty)`) directly as fields, 1-place predicates (e.g. `(ontable ?x)`) as `BitVector`s, and 2-place predicates (e.g. `(on ?x ?y)`) as `BitArray`s). Since bit arrays are highly compact, this leads to significantly reduced memory use and allocation overhead. In addition, PDDL.jl generates fast accessor implementations for compiled state representations, directly looking up the appropriate index for a particular set of object arguments to a fluent, instead of performing a hash computation each time. In the Blocksworld example of Figure 5-2, these compilations lead to state representations which require about five times less memory (1720 vs. 336 bytes), and run about seven times faster when accessing the value of fluents (394 vs. 58.5 ns).

We note that this compilation strategy can readily be modified to use other array data types if they prove to be more efficient. For example, sparse arrays could be used for binary predicates over a large number of objects, statically-sized arrays can be used to speed up operations when the number of objects is small, and views of an underlying contiguous array could be used to further reduce overhead from memory allocation. In the future, representing mutually exclusive predicates as finite-domain variables [76] may lead to even greater improvements in speed and memory use.

5.2 Compiled Action Semantics

In addition to compiled state representations, PDDL.jl also supports *compiled action semantics*, generating specialized implementations of the `execute` and `available` interface methods for each action schema in the compiled domain. This makes use of Julia's support for multiple dispatch: By generating concrete subtypes of the `Action` datatype for each action schema, specialized methods can be defined for each subtype.

```
function execute(domain, state, action::GenericAction, args)
    # Substitute arguments into effect formula
    subst = Subst(var => val for (var, val) in zip(get_argvars(action), args))
    effect = substitute(get_effect(action), subst)
    # Interpret effect formula and return a state difference
    diff = effect_diff(domain, state, effect)
    # Apply state difference
    state = update(domain, state, diff)
    return state
end
```

(a) Generic implementation of `execute` for uncompiled actions (Runtime: 21.1 µs)

```
function execute(domain, state, action::CompiledStackAction, args)
    state = copy(state)
    # Get object indices for arguments
    x_idx = objectindex(state, :block, args[1].name)
    y_idx = objectindex(state, :block, args[2].name)
    # Assign new values to affected fluents
    state.handempty = true
    state.clear[x_idx] = true
    state.clear[y_idx] = false
    state.holding[x_idx] = false
    state.on[x_idx, y_idx] = true
    return state
end
```

(b) Compiled implementation of `execute` for Blocksworld's `stack` action (Runtime: 337 ns)

Figure 5-3: Interpreted vs. compiled implementations of the `execute` method for applying the effects of an `Action` to a `State`.

An example is shown in Figure 5-3, which compares (a) the generic interpreter implementation of `execute` to (b) the compiled implementation of `execute` for the `(stack ?x ?y)` action in the Blocksworld domain. Instead of interpreting the effect formula associated with the action each time it is executed, the compiled version of

`execute` directly modifies the appropriate entries of the compiled state representation for the Blocksworld domain. It does this by first looking up the array indices that correspond to each argument of the action using the `objectindex` method (which is itself compiled), and sets the value of the affected entries in accordance with the add, delete, and assignment effects specified by the PDDL action schema. All of this code is compiled from PDDL to Julia, which in turn gets compiled to high performance machine code by Julia's own compiler. By directly modifying the state representation, instead of performing substitution and interpretation of the PDDL syntax tree each time, the compiled implementation in Figure 5-3 achieves median runtimes up to 60 times faster (336 ns vs. 21.1 µs) than the generic implementation of `execute`. The tradeoff is that code generation and compilation takes time, leading to initial overhead cost that may not always be favorable relative to the interpreter, depending on the downstream context.

5.3 Static Analysis and Other Compiler Techniques

Static analysis of PDDL domains and problems can be used to derive useful information in advance of planning, such as the presence of static fluents (i.e. "fluents" that are never modified by actions), the reachability of ground actions (i.e. whether they ever become applicable), the set of predicates that are action-relevant (i.e. necessary to determine some action's availability), or the set of predicates that are goal-relevant (i.e. may appear in a plan to achieve the goal). This information can then be used to aid compilation as well as prune the search space. For example, by inferring the set of static fluents, compiled state representations can omit storing the values of those fluents as mutable variables, reducing memory use. Furthermore, ground actions can be eliminated as unavailable because they have static preconditions that are never satisfied, thereby reducing the search space.

PDDL.jl currently supports these analyses as a set of user-available tools, which can be used in the context of planning algorithms or heuristics to reduce the search space and improve performance. These tools will be eventually used by the PDDL.jl

compiler to perform some of the space and time saving compilations mentioned above. In addition, PDDL.jl supports *domain grounding*, i.e. generating the set of all possible ground actions and fluents. This is one of the most widely used static transformations by symbolic planning systems to accelerate planning, because it enables the implementation of a variety of useful static analyses (e.g. determining the unreachability or irrelevance of ground actions, which can then be pruned), and also avoids the need to dynamically ground actions during search. Notably, PDDL.jl supports this grounding transformation and associated analyses for domains that make use of quantified expressions and conditional effects allowed by the ADL extension of PDDL [41], as well as numeric fluents introduced in PDDL 2 [45]. This distinguishes PDDL.jl from widely used planning systems such as Pyperplan [38], which does not support ADL, or FastDownward [17], which does not support numeric fluents.

A number of other advanced compilation techniques can be implemented with the help of static analysis. Two such techniques used in state-of-the-art planning systems include *successor generators*, a decision tree-like data structure that allows for the rapid determination of the set of available actions, or *finite-domain representations*, which require inferring which sets of ground predicates are mutually exclusive and can hence be transformed into finite-domain variables [17]. While PDDL.jl does not yet implement these techniques, the fact that we abstract its interface from its (multiple) implementations means that future improvements to the compiler can be made with relative ease. As such, we anticipate that these advanced compilation techniques will eventually be supported by PDDL.jl as well.

5.4 Extended Compiler Semantics

In Chapter 3, we described how the PDDL.jl interpreter can be modified to support extended PDDL semantics via custom functions, semantic theories, and imperatives. Apart from custom imperatives, this extensibility applies to the built-in compiler as well. This is because Julia is the target language for the compiler, and the interface for specifying custom functions or theories requires implementations as Julia functions

and data types. As such, when compiling PDDL expressions that contain custom functions (e.g. in preconditions or effects), the compiler can automatically look up and insert their Julia implementations within the generated code. Compiled state representations can also be extended in a similar way, provided that users include type annotations in the PDDL domain file for fluents with custom data types (e.g. the set-valued fluents in Figure 3-4). These PDDL fluent types are then mapped to their Julia type implementations when generating a compiled state.

This implementation strategy means that the *abstract* semantics introduced in Chapter 4 can also be compiled. In particular, because abstract interpretation is implemented by introducing new Julia data types to represent abstract values (e.g. the interval abstraction over numeric fluents) then overloading the Julia implementations of PDDL functions (e.g. `>` or `+`) to operate over these new types, the compiler can support abstract semantics simply by generating compiled state representations that use these abstraction types. For example, if the compiled state contains an interval-valued numeric fluent called `fuel` represented by the `IntervalAbs` data type, then in the course of executing a functional expression such as `(> fuel 0)`, Julia will look up and dispatch to the method for `>` that is defined over an `IntervalAbs` and `Int` argument, which will perform the appropriate interval arithmetic. These compiled abstract semantics can be supported for custom theories as well: Users just need to define a data type for abstract values in the theory of interest, then overload the Julia implementations of custom functions to work over the new type.

Given the extensibility of the compiler, it is worth asking what functional limitations it might still have relative to the PDDL.jl interpreter. As of the current version of PDDL.jl, these limitations are (i) the inability to handle custom imperatives; (ii) lack of support for reverse execution; and (iii) restrictions on Julia that prevent fully dynamic code generation for compiled states and actions (which may be desirable when the number of objects is not fixed). As such, the interpreter is still strictly more general in the functionality it supports, while also being more easily debugged. In the future, however, improvements to the compiler could narrow this functionality gap, such that the interpreter will primarily be useful for debugging purposes.

Chapter 6

Applications and Evaluation

The preceding chapters of this thesis have described several features of PDDL.jl that allow it serve as general, extensible, and performant architecture for symbolic planning applications. In this chapter, we demonstrate and evaluate the utility of this architecture in the context of several downstream applications. To illustrate the generality of PDDL.jl, we show how it can be used within a wide variety of such applications: (i) symbolic planning algorithms, (ii) environment simulation for reinforcement learning, (iii) symbolic state estimation from noisy or partial observations, and (iv) goal inference via Bayesian inverse planning. These applications also exhibit the extensibility of PDDL.jl, both in terms of extended domain semantics and the ease with which PDDL.jl can be composed with existing libraries in the Julia ecosystem for reinforcement learning or probabilistic programming. In addition, we evaluate the performance of PDDL.jl when used within these applications, comparing against relevant baselines where possible.

6.1 Planning Algorithms

The most direct application of PDDL.jl is the implementation of symbolic planning algorithms. As illustrated by the pedagogical examples in Section 2.4, a variety of such algorithms can be implemented using the PDDL.jl interface, including forward and regression planners. In this section, we introduce a library of such algorithms

	Automated Planning Systems					
	Symbolic Planners.jl	Pyperplan [38]	PDDL4J [39]	HSP [15]	FastDownward [17]	ENHSP [21, 22]
Language Support						
STRIPS	✓	✓	✓	✓	✓	✓
ADL	✓		✓	✓	✓	✓
Numeric Fluents	✓					✓
Sets, Arrays, etc.	✓					
Events & Processes						✓
Planning Algorithms						
Forward Search	✓	✓	✓	✓	✓✓	✓
Regression Search	✓			✓		
Decision-Theoretic	✓					
Heuristic Coverage						
Propositional Domains	✓	✓	✓	✓	✓✓	✓
Numeric Domains	✓					✓✓
Sets, Arrays, etc.	✓					

Table 6.1: Comparison of features currently supported by SymbolicPlanners.jl versus other planning systems. A ✓ indicates support, while ✓✓ denotes highest coverage (most algorithms implemented, etc.) among systems that support the feature.

written using PDDL.jl called SymbolicPlanners.jl [32], and evaluate the performance of a subset of these algorithms.

SymbolicPlanners.jl is a library that provides a suite of planning algorithms and heuristics that operate domains and problems specified in PDDL. Similar to PDDL.jl, the library defines an abstract interface for heuristics, planners, their corresponding solutions (e.g. plans or policies), as well as goal, reward, and constraint specifications. Implemented planners currently include forward state space planners (breadth-first search, greedy best-first search, A*, etc.) [15], regression planners (backward A*, etc.) [77], and decision-theoretic planners such as RTDP [3] and MCTS [78]. Supported heuristics include Manhattan or Euclidean distance heuristics for user-specified fluents, standard delete-relaxation heuristics such as $h_{\max}$, h_{add} and h_{FF}, as well as extensions of these heuristics to work with numeric or functional fluents. In contrast to most contemporary planning systems, SymbolicPlanners.jl has broader language support for variants and extensions of PDDL, includes families of planning algorithms beyond forward state space search, and provides heuristics that generalize to domains with non-Boolean fluents. A comparison of supported features is shown in Table 6.1.

In addition to supported features, we compare the performance of SymbolicPlanners.jl against several widely used planning systems:

- **Pyperplan** [38], a lightweight STRIPS planner written in Python that is often used in contemporary planning research due to its ease of modification (e.g. to integrate machine learning and reinforcement learning [29, 79, 80, 81]),
- **FastDownward** [17], a state-of-the-art planning system written in C++ that operates over propositional domains,
- **ENHSP** [21, 22], a heuristic search planner for numeric domains written in Java, which extends the $h_{\max}$ and h_{add} heuristics to support numeric conditions.

To evaluate performance, we collected runtimes and other statistics for each system on a variety of planning domains featured in the 2000 and 2002 International Planning Competitions [82, 83]:

- Blocksworld, Logistics, and Miconic (ADL) for propositional planning.
- Zeno Travel, Depots, and Rovers for numeric planning.

Pyperplan was only evaluated on the Blocksworld and Logistics domains due to its lack of support for ADL. FastDownward was evaluated on all propositional domains, while ENHSP was evaluated on the numeric domains. SymbolicPlanners.jl was evaluated on all domains, using both the PDDL.jl interpreter and compiler. All evaluation runs were performed on an Intel Core i7-8665U CPU @ 2.11 GHz with 16.0 GB of RAM with a 64-bit Windows 10 Pro operating system, with three runs per problem, and a time limit of 180 seconds per run. Each planning system was configured to use the same planning algorithm and heuristic (or the closest possible), so as to evaluate the performance difference arising from implementational efficiency. To investigate the impact of heuristic guidance versus implementation efficiency, we also conducted a two-way comparison within SymbolicPlanners.jl, altering both the informativity of the search heuristic and the implementation used (interpreter vs. compiler) to assess the contribution of each to high-performance planning.

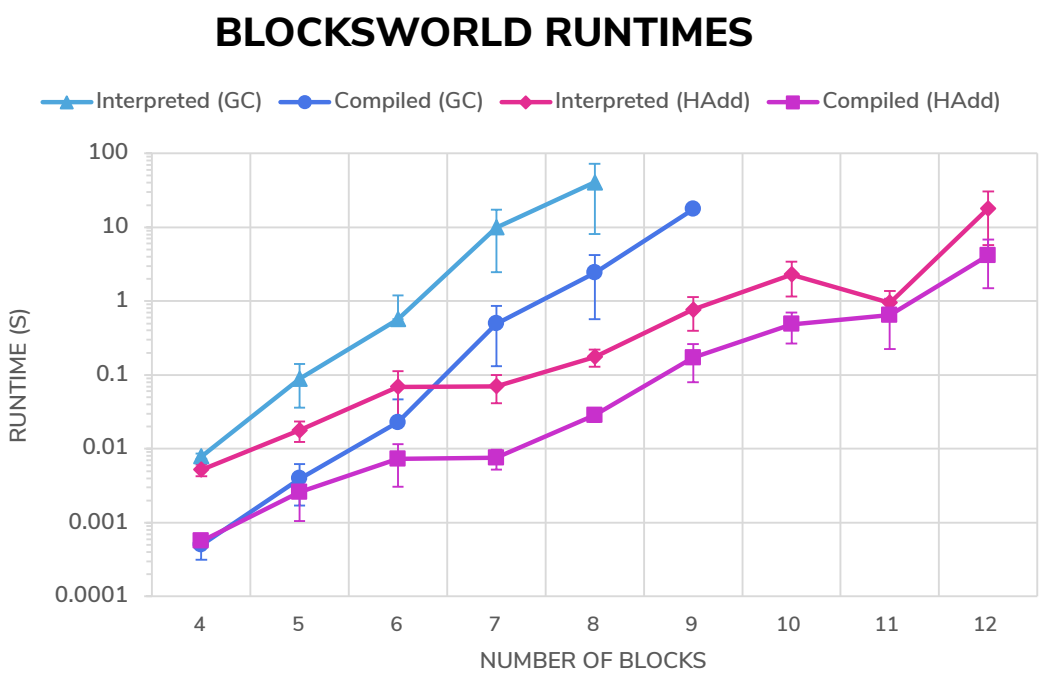


Figure 6-1: Runtime vs. problem size (number of blocks) for Blocksworld problems across planning heuristics and PDDL.jl implementations. GC refers to the uninformative goal count heuristic. HAdd refers to the h_{add} heuristic. A* search was used in all cases. Error bars show standard deviations across runs for each problem size.

Figure 6-1 compares runtimes across different planning heuristics and PDDL.jl implementations in the Blocksworld domain, while Table 6.2 summarizes this comparison across all propositional domains. As can be seen from Figure 6-1, compilation leads to a constant factor improvement in runtime regardless of planning heuristic. In contrast, the choice of planning heuristic strongly affects the rate of exponential growth of runtime as a function of problem size — using an informative heuristic such as h_{add} leads to much slower growth in search complexity relative to the uninformed goal count heuristic (which counts the number of unsatisfied sub-goals). These trends hold across domains, as shown by Table 6.2: Compilation improves median runtime by about one order of magnitude, while using the goal count heuristic is one to three orders of magnitude slower than h_{add}. This is due to the much larger number of nodes expanded during search when using the goal count heuristic, which has a log node expansion ratio of 4 to 7 relative to h_{add}, indicating rapid exponential growth.

Relative Runtime: Q1 | M | Q3				
	h_{add}		Goal Count	
Domain	**Compiled**	**Interpreted**	**Compiled**	**Interpreted**
Blocksworld	1.0 | 1.0 | 1.0	4.6 | 6.4 | 8.8 $\times 10^0$	0.1 | 0.7 | 6.0 $\times 10^1$	0.2 | 1.1 | 6.9 $\times 10^2$
Logistics	1.0 | 1.0 | 1.0	0.6 | 1.1 | 1.8 $\times 10^1$	0.2 | 0.5 | 1.5 $\times 10^3$	3.2 | 5.7 | 9.6 $\times 10^3$
Miconic	1.0 | 1.0 | 1.0	1.0 | 1.2 | 1.4 $\times 10^1$	0.1 | 0.5 | 4.9 $\times 10^2$	0.1 | 0.5 | 2.2 $\times 10^3$

Log Node Expansion Ratio: Q1 | M | Q3		
Domain	h_{add}	**Goal Count**
Blocksworld	0.0 | 0.0 | 0.0	1.8 | 5.0 | 5.9
Logistics	0.0 | 0.0 | 0.0	6.0 | 7.0 | 7.9
Miconic	0.0 | 0.0 | 0.0	2.1 | 4.2 | 6.8

Table 6.2: Runtime and log node expansions for A* search in SymbolicPlanners.jl using different heuristics and PDDL.jl implementations, measured relative to the compiled implementation using the h_{add} heuristic. In each cell, we report the first quartile (Q1), median (M), and third quartile (Q3) across solved problems.

When we fix the heuristic to h_{add} and compare performance across different planning systems using A* search, we find that SymbolicPlanners.jl is highly competitive with existing software. As Figure 6-2 shows, SymbolicPlanners.jl is consistently faster than Pyperplan in the Blocksworld domain. This is especially true when using the PDDL.jl compiler, which delivers more than an order of magnitude speed-up over Pyperplan. Compared to FastDownward, which represents the state-of-the-art, SymbolicPlanners.jl also performs competitively, achieving faster or equivalent runtimes when using the PDDL.jl compiler on smaller problems, and staying within an order of magnitude for larger problems.

These trends are again borne out across different domains, as shown by Table 6.3. On both the Blocksworld and Logistics domains, SymbolicPlanners.jl is between 15 to 23 times faster than Pyperplan in terms of median runtime. Across all propositional domains, it is only about 3 times slower than FastDownward. Given that SymbolicPlanners.jl is considerably more general than both Pyperplan and FastDownward in terms of features and language support, these results imply that it and PDDL.jl can replace Pyperplan as a more performant high-level platform for automated planning research, and also serve as a viable alternative to FastDownward, especially when some performance is worth sacrificing for generality and extensibility.

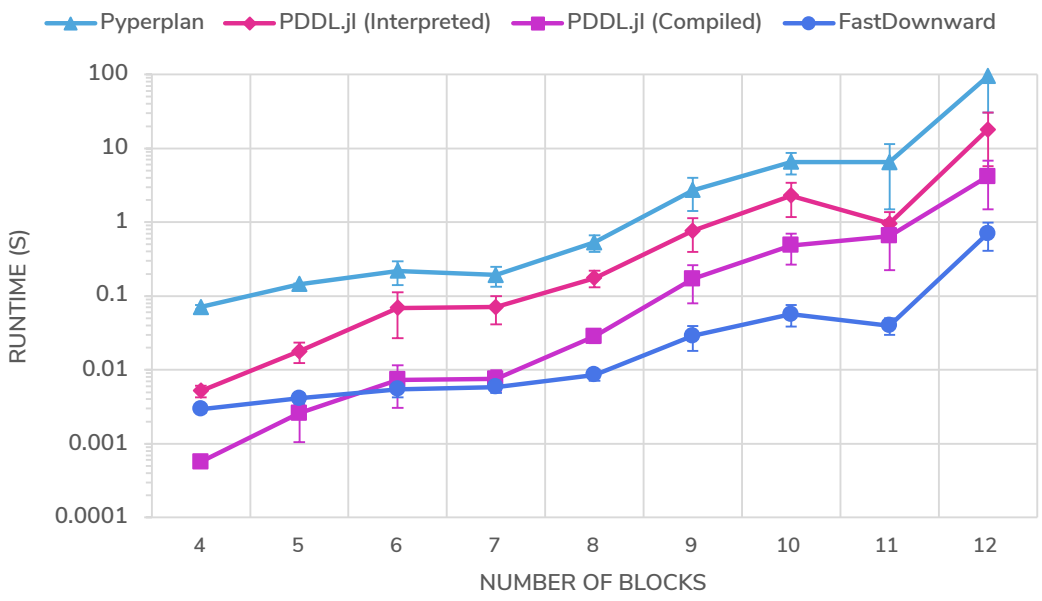


Figure 6-2: Runtime vs. problem size (number of blocks) for Blocksworld problems, compared across Pyperplan, FastDownward, and SymbolicPlanners.jl using both the PDDL.jl interpreter and compiler. A* search with the h_{add} heuristic was used in all cases. Error bars show standard deviations across runs for each problem size.

	Relative Runtime: Q1 | M | Q3 *(Fraction Solved)*				
	SymbolicPlanners.jl		**Pyperplan**	**FastDownward**	**ENHSP**
Domain	Compiled	Interpreted	[38]	[17]	[21, 22]
Blocksworld	1.0 | 1.0 | 1.0 *(26 / 26)*	4.6 | 6.4 | 8.8 *(26 / 26)*	19. | 23. | 53. *(26 / 26)*	0.2 | 0.3 | 1.2 *(26 / 26)*	—
Logistics	1.0 | 1.0 | 1.0 *(23 / 24)*	6.4 | 11. | 18. *(23 / 24)*	10. | 15. | 23. *(23 / 24)*	0.1 | 0.3 | 0.5 *(23 / 24)*	—
Miconic	1.0 | 1.0 | 1.0 *(150 / 150)*	10. | 12. | 14. *(150 / 150)*	—	0.1 | 0.3 | 0.6 *(150 / 150)*	—
Zeno Travel	1.0 | 1.0 | 1.0 *(13 / 15)*	8.7 | 9.8 | 12. *(13 / 15)*	—	—	5.0 | 7.0 | 36. *(13 / 15)*
Depots	1.0 | 1.0 | 1.0 *(8 / 10)*	8.7 | 9.3 | 9.5 *(4 / 10)*	—	—	0.7 | 1.1 | 8.1 *(8 / 10)*
Rovers	1.0 | 1.0 | 1.0 *(6 / 10)*	19. | 22. | 27. *(4 / 10)*	—	—	2.5 | 2.5 | 2.5 *(1 / 10)*

Table 6.3: Relative runtime and fraction of problems solved within the time limit, compared across different planning systems and domains. Runtimes are relative to SymbolicPlanners.jl using the PDDL.jl compiler. In each cell, we report the first quartile (Q1), median (M), and third quartile (Q3) across solved problems.

One important way in which SymbolicPlanners.jl is more general than FastDownward and Pyperplan is its support for numeric domains. Table 6.3 also summarizes its performance on such domains, comparing against ENHSP as a baseline. Specifically, we use ENHSP-20 with A* search and its generalization of the h_{add} heuristic to support numeric subgoals [22]. For SymbolicPlanners.jl, we use a similar but simpler generalization of h_{add} to numeric domains which avoids the need for interval-based relaxations (supported by the PDDL.jl abstract interpreter (Chapter 4), but currently not optimized for use in a h_{add}-style heuristic) at the cost of some informativeness. We also ignore non-unit action costs, using each system to find a plan that satisfices rather than optimizes. The results show that SymbolicPlanners.jl generally outcompetes ENHSP in this setting, solving all problems that ENHSP is capable of solving before timing out, while achieving median runtimes that are 1.1 to 7.0 times faster. On the Rovers domain in particular, SymbolicPlanners.jl is able to solve 6 out of 10 problems within the time limit, whereas EHNSP only solves 1 out of 10, though this may partly be due to the choice of heuristic and planning algorithm.

The results described in this section demonstrate that using PDDL.jl for symbolic planning is a compelling alternative to existing systems in terms of both generality and performance. However, the utility of PDDL.jl is not limited to planning algorithms alone. In the following sections, we show how PDDL.jl can be flexibly composed with other libraries via its interface, enabling a wide range of contemporary AI applications.

6.2 Environments for Reinforcement Learning

Reinforcement learning (RL) methods have become highly popular in recent years, in part due to their ability to achieve super-human performance on complex sequential decision-making problems via integration with deep neural networks [84, 85, 86]. However, the time required to train such (typically model-free) methods is often many orders of magnitude greater than the time required for planning by model-based methods [87], such as the heuristic search algorithms considered in Section 6.1. To facilitate comparisons between each of these approaches, and to enable the de-

```
"Construct a symbolic MDP from a PDDL domain and problem."
function SymbolicMDP(domain::Domain, problem::Problem)
    state = PDDL.initstate(domain, problem)
    goal = PDDL.get_goal(problem)
    metric = PDDL.get_metric(problem)
    if metric !== nothing # Extract metric formula to minimize
        metric = metric.name == :minimize ?
            metric.args[1] : Compound(:-, metric.args)
    end
    return SymbolicMDP(domain, state, goal, metric)
end

POMDPs.initialstate(m::SymbolicMDP) =
    Deterministic(m.init)
POMDPs.actions(m::SymbolicMDP, s) =
    PDDL.available(m.domain, s)
POMDPs.transition(m::SymbolicMDP, s, a) =
    Deterministic(PDDL.transition(m.domain, s, a))
POMDPs.reward(m::SymbolicMDP, s, a, sp) = m.metric === nothing ?
    -1 : PDDL.evaluate(domain, s, m.metric) - PDDL.evaluate(domain, sp, m.metric)
POMDPs.discount(m::SymbolicMDP) =
    m.discount
POMDPs.isterminal(m::SymbolicMDP, s) =
    PDDL.satisfy(m.domain, s, m.goal)
```

Figure 6-3: SymbolicMDPs.jl, a MDP and RL environment interface for PDDL domains, can be easily implemented by wrapping the PDDL.jl interface within the POMDPs.jl package due to their close correspondence.

velopment of hybrid planning algorithms that combine their advantages [28, 29, 81], it is thus helpful to encapsulate symbolic PDDL domains and problems within RL environments, allowing them to be used with contemporary RL algorithms.

PDDL.jl supports this functionality, because its interface is easily composed with existing frameworks for RL and Markov decision processes (MDPs). As a demonstration, we present SymbolicMDPs.jl [34], a library which provides exactly this functionality by integrating PDDL.jl with the POMDPs.jl library for partially observable MDPs [88], allowing a wide range of MDP solvers to be applied to PDDL domains. Since POMDPs.jl interoperates with the ReinforcementLearning.jl library for RL environments and algorithms [89], as well as the AlphaZero.jl library for neurally-guided MCTS [90, 91], this also means that a plethora of modern deep RL algorithms can be tested on PDDL problems and compared against symbolic planning algorithms. Figure 6-3 shows an excerpt of the source code of SymbolicMDPs.jl. The PDDL.jl

interface closely mirrors the POMDPs.jl interface and the underlying concepts that define MDPs. As a result, most of the implementation is extremely straightforward, consisting of one-line mappings between POMDPs.jl and PDDL.jl methods.

SymbolicMDPs.jl is similar to and inspired by PDDLGym [92], a Python library that allows PDDL domains and problems to be used as OpenAI Gym environments [93] for reinforcement learning. We thus use it as a baseline for comparison. We omit a full feature comparison as both libraries remain in development, noting only that while PDDLGym benefits from easier integration with machine learning algorithms in the Python ecosystem, it does not currently support some language features that SymbolicMDPs.jl inherits from PDDL.jl, especially the ability to handle numeric fluents and conditional effects. For performance, we compare the frames per second (FPS) that a PDDL-based RL environment can be simulated at, since fast simulation means learning can occur at a faster pace. Results are shown in Table 6.4 for a representative sample of domains supported by PDDLGym. On most domains, SymbolicMDPs.jl with the PDDL.jl compiler is the fastest, though on domains such as Sokoban and Depot, the PDDL.jl interpreter or PDDLGym is faster. This is likely due to the dynamic action grounding strategy used by the interpreter being considerably faster when there are large numbers of possible ground actions, as in Sokoban. Future improvements to the compiler (Section 5.3) may reverse this trend.

	Frames Per Second		
	SymbolicMDPs.jl		**PDDLGym**
Domain	Compiled	Interpreted	Sep 2020 [92]
Meet-Pass	**32787**	16260	7380
Blocksworld	**25316**	4587	7064
Baking	4219	1356	**5897**
Hanoi	**22472**	12579	4580
USA Travel	**3846**	27	1251
Doors	1034	**1812**	917
Sokoban	3	**1190**	155
Depot	41	38	**97**

Table 6.4: Frames per second (FPS) for eight PDDL-based RL environments, compared across SymbolicMDPs.jl and PDDLGym. Results for PDDLGym are from [92]. For SymbolicMDPs.jl, we adopt a similar evaluation procedure, measuring average FPS for a domain by running a random policy for 10 timesteps over 200 episodes.

We conclude this section with a brief survey of RL algorithm research that SymbolicMDPs.jl might enable. Since environment states in SymbolicMDPs.jl support the same interface defined by PDDL.jl, we can query the value of relations between objects in the state, enabling deep relational reinforcement learning [94, 28] by converting states into representations suitable for graph and hypergraph neural networks [95, 79, 96]. In addition, since we can easily specify goals in terms of logical formulae, this enables research into goal-driven curiosity learning via goal sampling [87, 97]. Finally, because domain-general heuristics can be used as estimates of the cost to reach a goal, they can be integrated with RL algorithms to either initialize value function estimators, or serve as proxy rewards [81]. By tightly integrating the engineering stack required for both symbolic planning and reinforcement learning, we believe that PDDL.jl and SymbolicMDPs.jl will be a significant aid to AI researchers pursuing these and many other directions.

6.3 State Estimation from Partial Observations

Planning in the real world requires reasoning under uncertainty. Details of the world relevant to planning are often only partially observable, and sensors are often imperfect, leading to noisy observations. Contending with this uncertainty is thus a key challenge for real world deployments of automated planning, such as in household robots and autonomous driving. Indeed, a traditional pitfall of classical approaches to planning is that they assume determinism in their symbolic models of the world, leading to brittleness when that deterministic assumption proves wrong.

While PDDL.jl inherits the legacy of classical AI, it is not beholden to its limitations when faced with uncertainty. In addition to supporting probabilistic extensions (Section 2.5.2), it can also be integrated with modern AI systems for reasoning about uncertainty, such as probabilistic programming libraries [98]. In this section, we show how PDDL.jl can be composed with Gen, a probabilistic programming system that provides a high-level interface for customizable inference [99], enabling estimation of PDDL states from partial or noisy observations in a principled Bayesian manner.

```
@gen function bw_observe(state::State)
    for x in PDDL.get_objects(state, :block)
        # Look up whether block is clear
        is_clear = PDDL.get_fluent(state, pddl"(clear $x)")
        # Sample observed value of clear with high probability
        {:clear => x.name} ~ Gen.bernoulli(is_clear ? 0.9 : 0.1)
    end
end
```

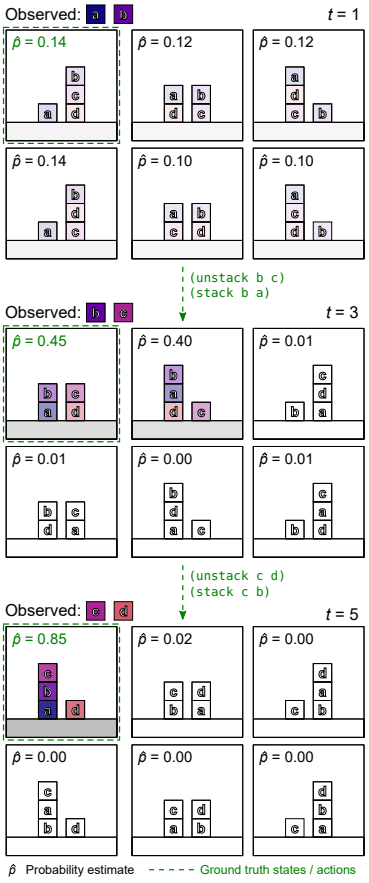
(a) Observation model

```
@gen function bw_model(domain::Domain, N::Int, T::Int)
    # Sample initial state and observation
    state ~ bw_prior(domain, N)
    {:obs => :init} ~ bw_observe(state)
    for t in 1:T
        # Sample and execute a random available action
        act ~ Gen.labeled_uniform(PDDL.available(domain, state))
        state = PDDL.transition(domain, state, act)
        # Sample observations given current state
        {:obs => t} ~ bw_observe(state)
    end
end
```

(b) State space model

```
function bw_inference(domain, n_blocks,
                      n_particles, observations)
    # Initialize filter with samples from model
    pf = pf_initialize(bw_model, (domain, n_blocks, 0),
                       observations[1], n_particles)
    for t in 1:length(observations)-1
        # Resample particles according to normalized weights
        pf_resample!(pf)
        # Increment timestep, condition on next observation
        pf_update!(pf, (domain, n_blocks, t),
                   (NoChange(), NoChange(), UnknownChange()),
                   observations[t])
    end
    return pf
end
```

(c) Particle filter for state estimation



(d) Inferred states over time

Figure 6-4: State estimation over a partially observable Blocksworld environment using PDDL.jl and Gen, specified by combining a PDDL domain (Figure 1-1) with (a) an observation model where only the top-most (i.e. clear) blocks are visible, and (b) a state space model. Inference is performed via (c) a particle filter written with Gen's building blocks, producing (d) an inferred distribution over state trajectories.

Figure 6-4 shows how to perform state estimation from a sequence of partial observations of Blocksworld states (Figure 1-1) using PDDL.jl and Gen. In this example, we make the simplified assumption that only the top-most block of any tower is observable (i.e., as if the observer is looking at the scene from top-down), mimicking the occlusion relations that exist in real 3D settings. We specify this observation model in Figure 6-4(a) as a Gen probabilistic program (`bw_observe`), which takes as input a PDDL.jl `State`, iterates over each block `?x` in the state, and samples an observed value of the predicate `(clear ?x)` that is correct with 90% probability (due to e.g. sensor noise). The predicate `(clear ?x)` indicates whether

a block has no other block on top of it, and hence is observable. Values of other fluents are not sampled, because they are assumed to be unobservable. Because PDDL.jl exposes fluent values through the `get_fluent` method (Definition 2.2.2), we can easily define observation models over PDDL states and their fluents in this way.

Having specified the observation model, we integrate this into a full state space model (`bw_model`) in Figure 6-4(b), which includes a prior (`bw_prior`) over initial Blocksworld states, and a transition distribution between states. For the prior, we assume we know the total number of blocks n, and sample an initial state uniformly at random using the algorithm given by Slaney and Thiébaux [100]. For the transition model, we sample an available action uniformly at random and execute it on the current state. Finally, we can perform inference over this model using the particle filtering algorithm shown in Figure 6-4(c). We initialize the filter by sampling a set of particles from the prior over initial states, then iteratively update the filter by incrementing the timestep, reweighting particles by the likelihood of new (partial) observations at that timestep, and resampling particles according to their normalized weights. Figure 6-4(d) shows this inference process in action over a sequence of partial observations of a Blocksworld state with four blocks, using 1500 particles. At $t = 1$, only block `a` and `b` are visible, so the particle filter estimates similar probabilities for all six states that are consistent with the observations (i.e. where the observed blocks are at the top of a tower). With more observations, the particles converge towards two states ($t = 3$) and then one state ($t = 5$) as the most likely possibility, due to the blocks newly observed as they are stacked and unstacked.

We note that the particle filtering algorithm described above is the most basic variant of sequential importance resampling [101], using the model itself as the proposal distribution. Even so, we achieve rapid inference, with each timestep t requiring approximately 0.06 seconds when using the PDDL.jl compiler with Gen, and sampling 1500 particles. However, as the number of blocks increases, the exponential increase in possible initial states and trajectory branches means that sampling from the prior may no longer be tractable. Fortunately, Gen also allows us to initialize and update our particle filter using custom proposal distributions specified as prob-

abilistic programs [102]. For example, we could initialize the filter using a program that biasedly samples Blocksworld states consistent with the initial observations. We could also propose actions between states that are more likely given the observations (e.g. if block `b` is visible in one state, and `c` in the next, this is likely due to block `b` being unstacked from `c`). This ability to customize inference means that inference algorithms can be easily be tailored for particular domains and application contexts.

Directly estimating PDDL states in this way has many downstream applications. One immediate application is that it allows symbolic planners to be extended to partially observable environments, while avoiding the need for hand-implemented belief-space representations and inference algorithms often used in robotics research [103, 104]. By combining PDDL.jl with Gen or other probabilistic programming systems, we can take advantage of their interfaces and algorithms to infer variational or particle-based beliefs about the current state under a wide variety of partial observability conditions. These inferred distributions can then be combined with partially-observable planning algorithms, such as online replanning from sampled states [105], or heuristic-guided variants of MCTS [106]. Another use case is Bayesian goal inference over agents operating in PDDL-specifiable domains [30]. State estimation is a sub-problem of this complex inference task that combines planning algorithms and partially observable environment models, which we turn to in the next section.

6.4 Goal Inference via Bayesian Inverse Planning

Many symbolic planning algorithms were initially inspired by how humans solve problems, either by working backwards from a goal [1], or by thinking ahead from the current state of the world while guided by heuristics [107]. This suggests using them as *models* of human planning and decision-making, allowing for prediction or inference over how humans might think or act. Indeed, this is exactly the approach adopted by a line of research known as Bayesian inverse planning, where planning algorithms are incorporated within Bayesian models of goal-directed agents, allowing for posterior inference over their goals and plans from observations of their environment and ac-

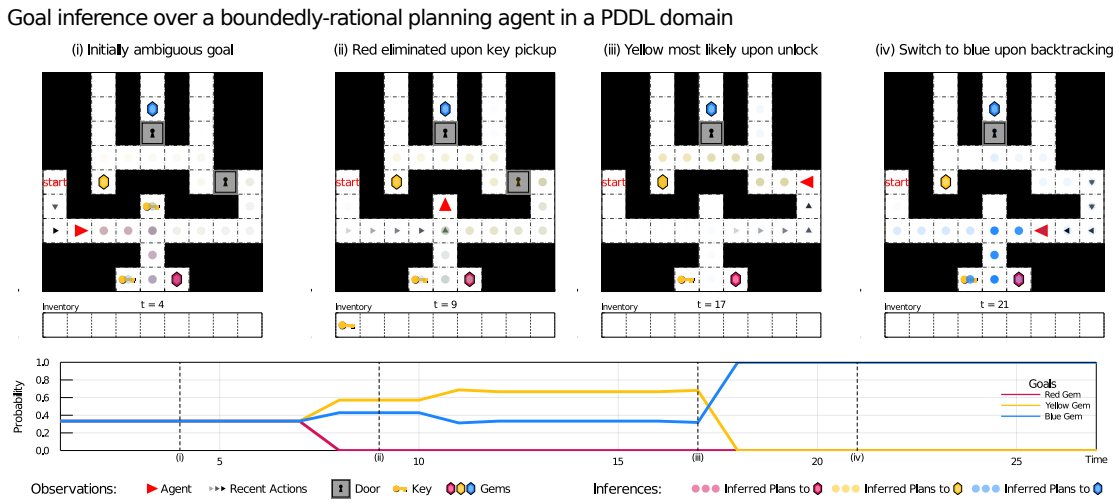


Figure 6-5: Visualization of goal inference in a PDDL domain, adapted from [30]. Using PDDL.jl with Gen, we can build models of agents that plan in PDDL domains, such as this gridworld with doors, keys and gems. From sequential observations of the environment, we can eventually infer the agent's true goal (the blue gem).

tions [108, 109, 110, 111]. Research in cognitive science has shown that this approach can account for how humans model and understand the actions of others, constituting a formal scientific framework known as Bayesian theory of mind [112, 113, 114]. Recent work by the author [30] has extended this approach to model boundedly-rational planning using Gen and an earlier version of PDDL.jl (Figure 6-5). In this section, we illustrate how PDDL.jl can be used to build such models, and how accelerated inference over these models is enabled by features of PDDL.jl presented in this thesis.

A key insight of recent approaches to Bayesian inverse planning is that (stochastic) planning algorithms can be viewed as probabilistic programs that define distributions over randomly generated plans [109, 110, 111]. As such, we can use off-the-shelf stochastic planners such as the RTDP and MCTS algorithms included in SymbolicPlanners.jl as modeling components, and perform sampling-based inference over their inputs (e.g. goals) and outputs (e.g. action sequences). Alternatively, planning algorithms can be directly specified as Gen programs to afford control over their internal random choices, or even automatically transformed into Gen using tools such as Genify [115]. In [30], we adopted the second approach, specifying a probabilistic variant of A* search as a Gen program that uses the PDDL.jl interface.

```
@gen function prob_astar(domain::Domain, state::State, goal::Term,
                         heuristic, search_noise, search_budget)
    # Initialize search tree and search queue
    tree = Dict(state => (distance=0, parent=state))
    queue = Dict(state => -heuristic(domain, state, goal))
    count = 0
    while !isempty(queue)
        # Sample next state from Boltzmann distribution over queue
        state ~ Gen.boltzmann(keys(queue), values(queue), search_noise)
        delete!(queue, state)
        count += 1
        # Return plan to sampled state if budget or goal is reached
        if count >= search_budget || PDDL.satisfy(domain, state, goal)
            return reconstruct_plan(state, tree)
        end
        # Otherwise iterate over successor states
        for act in PDDL.available(domain, state)
            next_state = PDDL.transition(domain, state, act)
            dist = tree[state].distance + 1
            # Update search tree and queue if shorter path is found
            if dist < tree[next_state].distance
                tree[next_state] = (distance=dist, parent=state)
                queue[next_state] = -(dist + heuristic(domain, state, goal))
            end
        end
    end
    return nothing
end
```

(a) Probabilistic A* search as a generative model in Gen using PDDL.jl

```
@gen function agent_env_model(domain::Domain, state::State, T::Int,
                              heuristic, search_noise, persistence)
    # Sample initial goal and observations
    goal ~ goal_prior()
    {:obs => :init} ~ obs_model(state)
    for t in 1:T
        if length(plan) < t
            # Sample search budget from negative binomial
            budget ~ Gen.neg_binom(persistence...)
            # Sample partial plan via probabilistic A* search
            part_plan ~ prob_astar(domain, state, goal,
                                   heuristic, search_noise, budget)
            append!(plan, part_plan)
        end
        # Execute action according to plan
        act = plan[t]
        state = PDDL.transition(domain, state, act)
        # Sample observations given current state
        {:obs => t} ~ obs_model(state)
    end
end
```

(b) Generative model of a planning agent interacting with a PDDL environment.

Figure 6-6: Generative models for Bayesian inverse planning using PDDL.jl and Gen. We model an agent that (a) plans using probabilistic A* search and (b) interacts with a PDDL environment over time to pursue an unknown goal. Conditioning on observations `{:obs => t}` of the environment allows us to infer the agent's goals.

Figure 6-6(a) shows a simplified version of this algorithm (`prob_astar`), highlighting again how PDDL.jl can be composed with Gen to build complex probabilistic models. As in standard A* search, we plan forward from the initial state by iteratively expanding neighboring states, guided by a heuristic (as provided by e.g. SymbolicPlanners.jl) that estimates the distance to the goal, until we reach the goal or use up a search budget. However, probabilistic A* samples the state to expand according to a Boltzmann distribution over the estimated path costs, such that more promising states are expanded more often, but not all the time. This mimics how human planning is generally efficient, but may not always be optimal.

We can incorporate this algorithm into the agent-environment model shown in Figure 6-6(b). Like the state space model in Figure 6-4(b), this model simulates the evolution of a PDDL environment over time, including potentially noisy observations defined by an observation model (`obs_model`). However, it also models the goals, plans and actions of an agent interacting with the environment. Initially, the agent pursues an unknown goal, which we sample from a prior (`goal_prior`). At each subsequent time step, the agent starts planning only if its existing plan does not extend to the current step, first sampling a search budget, then extending its plan with a partial plan generated via probabilistic A* search (`prob_astar`). The agent then executes an action according to its plan, which changes the environment's state per the transition function specified by the PDDL domain. Goal inference can be performed by conditioning on observations of the environment's state, using a particle filtering algorithm similar to Figure 6-4(c) called Sequential Inverse Plan Search (SIPS) [30]. An example of goal inference on a PDDL domain called Doors, Keys & Gems is illustrated in Figure 6-5. In this domain, the agent's goal is to collect one of three colored gems. The inferred distribution over goals is initially uncertain ($t = 4$), but converges towards the blue gem as other possibilities are eliminated ($t = 21$).

Since our key focus here is on how PDDL.jl enables applications such as goal inference, we avoid further exposition of the SIPS algorithm and refer readers to [30] and [114] for technical details and motivations. The only feature of SIPS we highlight is that, as a particle filtering algorithm, it benefits from the ability to efficiently

simulate more particles (a.k.a. samples), where each particle corresponds to a possible way the agent-environment model (Figure 6-6(b)) could evolve: With more particles, more accurate estimation of the posterior distribution is possible [116]. This in turn means that faster planning algorithms and PDDL implementations should enable more accurate and efficient inference by reducing the runtime cost for each particle.a

To evaluate whether PDDL.jl delivers these benefits, we reproduce the quantitative goal inference experiments on the Doors, Keys & Gems and Block Words domains from [30] with a few adjustments. In addition to testing the older version of PDDL.jl (v0.1) and associated code from the original experiments, we re-run the experiments using the updated version of PDDL.jl (v0.2) presented in this thesis, evaluating both the interpreter and compiler. For the Doors, Keys & Gems domain, we update the representation to use compact array-valued fluents as a custom semantic theory (Section 3.2). For the Block Words domain (a version of Blocksworld where goals correspond to words spelled from lettered blocks), we also use a faster implementation of the h_{add} heuristic from SymbolicPlanners.jl (Section 6.1). Finally, we adjust the number of samples used by the particle filter, using either 10 samples or 50 samples per goal to investigate the impact on speed and accuracy.

Domain	**Configuration**			**Accuracy** (P_{true})			**Runtime** (s)		
(n Goals)	PDDL.jl	Compiled	Samples	Q1	Q2	Q3	IC	MC	AC
Doors, Keys & Gems (3 Goals)	v0.1	No	30	**0.38**	0.50	0.65	3.36	0.15	0.30
	v0.2	No	30	**0.37**	0.50	**0.68**	3.25	0.07	0.24
	v0.2	No	150	**0.38**	**0.53**	**0.67**	4.32	0.18	0.39
	v0.2	Yes	30	**0.37**	**0.51**	**0.66**	**0.41**	**0.01**	**0.03**
	v0.2	Yes	150	**0.39**	**0.52**	**0.67**	0.71	0.04	0.08
Block Words (5 Goals)	v0.1	No	50	0.46	0.79	0.82	10.3	0.96	1.82
	v0.2	No	50	**0.47**	0.80	0.84	1.63	0.16	0.30
	v0.2	No	250	**0.47**	**0.84**	**0.90**	5.85	0.88	1.34
	v0.2	Yes	50	**0.48**	0.82	0.86	**0.76**	**0.09**	**0.16**
	v0.2	Yes	250	**0.48**	**0.85**	**0.89**	2.00	0.39	0.54

Table 6.5: Accuracy and runtime metrics for goal inference over PDDL domains. Accuracy is measured as the posterior probability of the true goal P_{true} at the 1st, 2nd and 3rd quartiles (Q1–Q3) of each observed trajectory. Runtime is measured in seconds, split into initial cost (IC), marginal cost per timestep (MC), and average cost per timestep (AC). Reported values are means across a dataset from [30], further averaged over 5 repetitions. Bolded values are within the 95% CI of the best value.

Table 6.5 shows accuracy and runtime metrics for these experiments, conducted on the same system as described in Section 6.1. In terms of accuracy, we find as expected that both the older and current versions of PDDL.jl perform similarly when the number of samples is fixed. With 5 times as many samples, we generally do not see significant improvements on the Doors, Keys & Gems domain, but find 2% to 8% increases in Q2 and Q3 accuracy on Block Words. This might be due to Block Words having a larger goal and state space than Doors, Keys & Gems, such that more particles allow for better coverage during inference. In terms of runtime however, we find that the new version of PDDL.jl leads to considerable improvements. This is especially so for the compiled implementation. With the same sample count, inference on Doors, Keys and Gems is 10 times faster relative to PDDL.jl v0.1 (which does not support compilation), and 3.75 times faster even when using 5 times as many samples. We see similar speed-ups on Block Words, with inference running 11 times faster for given equal sample counts, and 3.4 times faster with 5 times more samples. Absolute runtime is low as well. Each environment timestep requires 0.03 to 0.54 seconds to process for the compiled implementation, which is faster than real-time if each step lasts on the order of seconds in the real world (e.g. a human stacking a block).

These improvements in runtime result from several contributions described in this thesis. On the Doors, Keys & Gems domain, array-valued fluents (Section 3.2) reduce the memory required to represent a gridworld state, which otherwise has to store a large number of Boolean fluents for every wall that is present. On the Block Words domain, a faster implementation of h_{add} is possible due to static analysis tools provided by PDDL.jl (Section 5.3), reducing initial planning cost. This is why even the interpreter for PDDL.jl v0.2 runs faster. Finally, compiled state representations (Section 5.1) and action semantics (Section 5.2) provide further constant factor speed-ups. Collectively, these features of PDDL.jl enable not only the construction of highly complex agent models, but also high performance inference over these models. It is our hope that these contributions will accelerate the development of such models and inference algorithms, thereby facilitating the creation of beneficial AI technologies which assist humans by accurately inferring their preferences, values and goals [117].

6.5 Other Applications and Future Work

Over the course of this thesis as a whole, we have explored the many ways in which PDDL.jl serves as a fast and flexible platform for applications of automated planning and many other connected areas of AI research. This final section summarizes some of the broad themes from that exploration, elaborating upon them in the context of potential applications and directions for future work.

The first theme is the value of well-designed interfaces, which carve the conceptual space of symbolic planning at pragmatically felicitous joints. While PDDL.jl makes no claim as to providing the optimal design for a symbolic planning interface, the modularity imposed by its interfaces is a key factor in enabling the many applications described in this chapter. Many of the contributions of the Chapters 3–5 also rely upon a clean separation between interface and implementation (interpreted, abstracted, or compiled). Indeed, questions about interface design surface in many of those chapters as well: how should the interpreter and compiler be designed to be extensible, and for abstractions to be customizable? By carefully providing solutions to these questions, PDDL.jl paves the way for a wide range of functionality extensions in the coming years. These include most of the extensions described in Section 2.5, all of which will go a long way towards bringing PDDL.jl to the "real world". Of these, supporting stochastic environments and task-and-motion planning will be especially useful for bringing PDDL.jl to robotics applications.

The second theme is the power of formal analyses and abstractions, for the purposes of both domain abstraction (Chapter 4) and domain compilation (Chapter 5). While this theme has largely faded from the limelight of AI research, the experiments in this chapter clearly demonstrate that both abstraction (in heuristics) and compilation have a significant impact on efficiency and performance. As discussed in the final sections of Chapters 4 and 5, there are many more ways to explore how abstractions and analyses could bring further performance improvements, or even deliver entirely new capabilities. One particularly exciting possibility is the use of abstract interpretation in the context of program synthesis for generalized plans. Programming

language researchers have spent many years exploring formal approaches to improving synthesis and search over the space of programs. The time is ripe for planning researchers to explore translating these ideas to the context of generalized planning, or perhaps vice versa. By providing a planning framework that prioritizes the ability to conduct and make use of formal and abstract analyses, we envision that PDDL.jl will enable the multifarious connections that exist between planning and programming to coalesce into concrete research.

The final theme is the richness of composition, integration, and hybridization of PDDL.jl with other technology platforms. While this echoes the first theme, our focus here is more upon the creative possibilities enabled by well-designed interfaces, as demonstrated by the latter three applications of this chapter. In each case, we showed that PDDL.jl flexibly composes with AI technology from a different subfield, allowing for the creation of applications beyond the reach of symbolic planning alone. To the extent that the next wave of AI technologies are likely to be hybrid applications such as these [26], this is a source of of tremendous value. The promise of composable platforms like PDDL.jl is that they can draw together disparate paradigms, toolkits, and research agendas, allowing them to be productively compared and combined in informative, ambitious, and exiting new ways. Just to list some possibilities, we noted earlier that SymbolicMDPs.jl naturally enables research into neuro-symbolic reinforcement learning, including model-free agents whose neural networks learn to directly approximate the value of relational states [28], but also model-based agents that combine state space search with neural heuristic learning [91]. On the Bayesian front, we could imagine using PDDL.jl with Gen for Bayesian program induction over PDDL domain theories, which could in turn be integrated into a model-based RL algorithm that combines model learning, neural heuristics, and explicit search. With advances in planning algorithms, we could also build ever richer agent models for Bayesian inverse planning, allowing assistive machines to better understand the complexity of our goals and intentions. These are but a sampling of the full-breadth of possibilities that PDDL.jl might enable. Whether it delivers on those possibilities shall be a question for future work.

Bibliography

[1] Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA, 1959.

[2] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[3] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003.

[4] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. STRIPS planning in infinite domains. *arXiv preprint arXiv:1701.00287*, 2017.

[5] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Integrated task and motion planning in belief space. *The International Journal of Robotics Research*, 32 (9-10):1194–1227, 2013.

[6] Fangkai Yang, Daoming Lyu, Bo Liu, and Steven Gustafson. PEORL: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. In *IJCAI*, 2018.

[7] Hubert L Dreyfus. Why Heideggerian AI failed and how fixing it would require making it more Heideggerian. *Philosophical psychology*, 20(2):247–268, 2007.

[8] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*, 2016.

[9] Pedro A Tsividis. *Theory-based learning in humans and machines*. PhD thesis, Massachusetts Institute of Technology, 2019.

[10] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - the Planning Domain Definition Language, 1998.

[11] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An introduction to the Planning Domain Definition Language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019.

[12] Håkan LS Younes and Michael L Littman. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*, 2:99, 2004.

[13] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 440–448, 2020.

[14] Dániel L Kovács and Tadeusz P Dobrowiecki. Converting MA-PDDL to extensive-form games. *Acta Polytechnica Hungarica*, 10(8):27–47, 2013.

[15] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[16] Jörg Hoffmann. FF: The Fast-Forward planning system. *AI magazine*, 22(3): 57–57, 2001.

[17] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[18] Mauro Vallati, Lukas Chrpa, Marek Grześ, Thomas Leo McCluskey, Mark Roberts, Scott Sanner, et al. The 2014 International Planning Competition: Progress and trends. *Ai Magazine*, 36(3):90–98, 2015.

[19] Jörg Hoffmann. The Metric-FF planning system: Translating"ignoring delete lists"to numeric state variables. *Journal of Artificial Intelligence Research*, 20: 291–341, 2003.

[20] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of artificial intelligence research*, 20:379–404, 2003.

[21] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez. Interval-based relaxation for general numeric planning. In *ECAI 2016*, pages 655–663. IOS Press, 2016.

[22] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez. Subgoaling techniques for satisficing and optimal numeric planning. *Journal of Artificial Intelligence Research*, 68:691–752, 2020.

[23] Miguel Ramírez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[24] Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic attachments for domain-independent planning systems. In *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[25] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

[26] Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: the 3rd wave. *arXiv preprint arXiv:2012.05876*, 2020.

[27] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4:265–293, 2021.

[28] Beomjoon Kim, Zi Wang, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning to guide task and motion planning using score-space representation. *The International Journal of Robotics Research*, 38(7):793–812, 2019.

[29] Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Learning neuro-symbolic relational transition models for bilevel planning. *arXiv preprint arXiv:2105.14074*, 2021.

[30] Tan Zhi-Xuan, Jordyn Mann, Tom Silver, Josh Tenenbaum, and Vikash Mansinghka. Online Bayesian goal inference for boundedly rational planning agents. *Advances in Neural Information Processing Systems*, 33, 2020.

[31] Naman Shah, Deepak Kala Vasudevan, Kislay Kumar, Pranav Kamojjhala, and Siddharth Srivastava. Anytime integrated task and motion policies for stochastic environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9285–9291. IEEE, 2020.

[32] Tan Zhi-Xuan. SymbolicPlanners.jl: Planning algorithms for problems and domains specified in PDDL., 9 2021. URL `https://github.com/JuliaPlanners/SymbolicPlanners.jl`.

[33] Peter Gregory, Derek Long, Maria Fox, and J Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[34] Tan Zhi-Xuan. SymbolicMDPs.jl: A MDP and RL interface for PDDL domains, 9 2021. URL `https://github.com/JuliaPlanners/SymbolicMDPs.jl`.

[35] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[36] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, 2010.

[37] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcis Palomeras, Natalia Hurtos, and Marc Carreras. Rosplan: Planning in the robot operating system. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, 2015.

[38] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan. `https://doi.org/10.5281/zenodo.3700819`, 2020. URL `https://doi.org/10.5281/zenodo.3700819`.

[39] Damien Pellier and Humbert Fiorino. PDDL4J: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176, 2018.

[40] Enrico Scala. PPMaJaL: A Java library to reason about PDDL. `https://gitlab.com/enricos83/PPMAJAL-Expressive-PDDL-Java-Library`, 2015.

[41] Edwin PD Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. *Kr*, 89:324–332, 1989.

[42] David E Smith. The case for durative actions: A commentary on PDDL2. 1. *Journal of Artificial Intelligence Research*, 20:149–154, 2003.

[43] Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In defense of PDDL axioms. *Artificial Intelligence*, 168(1-2):38–69, 2005.

[44] Maria Fox and Derek Long. PDDL+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34, 2002.

[45] Maria Fox and Derek Long. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[46] Scott Sanner and Craig Boutilier. Practical linear value-approximation techniques for first-order mdps. *arXiv preprint arXiv:1206.6879*, 2012.

[47] Eric A Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[48] Caelan Reed Garrett, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research*, 37(1):104–136, 2018.

[49] Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, 2015.

[50] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9883–9891, 2020.

[51] Dániel László Kovács. A multi-agent extension of PDDL3. 1. *WS-IPC*, page 19, 2012.

[52] Alfonso Gerevini and Derek Long. BNF description of PDDL3. 0. *Unpublished manuscript from the IPC-5 website*, 2005.

[53] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[54] Tan Zhi-Xuan. Julog.jl: A Julia package for Prolog-style logic programming, 3 2020. URL `https://github.com/ztangent/Julog.jl`.

[55] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *European Conference on Planning*, pages 1–20. Springer, 1999.

[56] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62: 535–577, 2018.

[57] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[58] León Illanes and Sheila A McIlraith. Numeric planning via abstraction and policy guided search. In *IJCAI*, pages 4338–4345, 2017.

[59] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4): 120–372, 2017.

[60] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *Aaai/iaai*, pages 119–125, 2002.

[61] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *The Knowledge Engineering Review*, 34, 2019.

[62] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning as heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 569–577, 2021.

[63] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[64] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 313–326, 2010.

[65] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL): 1–30, 2017.

[66] Andrew I Coles and Amanda J Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28: 119–156, 2007.

[67] Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In *The Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS), Breckenridge, Colorado, 14-17 April, 2000.*, pages 140–149. AAAI Press, 2000.

[68] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems*, pages 3–18. Springer, 2008.

[69] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM (JACM)*, 61(3):1–63, 2014.

[70] Luis Benet and David P. Sanders. IntervalArithmetic.jl: Rigorous floating-point calculations using interval arithmetic in Julia, January 2022. URL `https://doi.org/10.5281/zenodo.5888392`.

[71] Earl D Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial intelligence*, 5(2):115–135, 1974.

[72] Craig A Knoblock. Learning abstraction hierarchies for problem solving. In *AAAI*, pages 923–928, 1990.

[73] John Hughes. *Backwards analysis of functional programs*. University of Glasgow. Department of Computing Science, 1987.

[74] David Monniaux. Backwards abstract interpretation of probabilistic programs. In *European Symposium on Programming*, pages 367–382. Springer, 2001.

[75] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 31–36, 2014.

[76] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[77] Vidal Alcázar, Daniel Borrajo, Susana Fernández, and Raquel Fuentetaja. Revisiting regression in planning. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[78] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[79] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 574–584, 2020.

[80] Rushang Karia and Siddharth Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 8064–8073, 2021.

[81] Clement Gehring, Masataro Asai, Rohan Chitnis, Tom Silver, Leslie Pack Kaelbling, Shirin Sohrabi, and Michael Katz. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. *arXiv preprint arXiv:2109.14830*, 2021.

[82] Fahiem Bacchus. AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *AI Magazine*, 22(3):47–47, 2001.

[83] Derek Long and Maria Fox. The 3rd International Planning Competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[84] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[85] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[86] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[87] Pedro A Tsividis, Joao Loula, Jake Burga, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J Gershman, and Joshua B Tenenbaum. Human-level reinforcement learning through theory-based modeling, exploration, and planning. *arXiv preprint arXiv:2107.12544*, 2021.

[88] Maxim Egorov, Zachary N Sunberg, Edward Balaban, Tim A Wheeler, Jayesh K Gupta, and Mykel J Kochenderfer. Pomdps. jl: A framework for sequential decision making under uncertainty. *The Journal of Machine Learning Research*, 18(1):831–835, 2017.

[89] Jun Tian and other contributors. ReinforcementLearning.jl: A reinforcement learning package for the Julia programming language, 2020. URL `https://github.com/JuliaReinforcementLearning/ReinforcementLearning.jl`.

[90] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[91] Jonathan Laurent. Alphazero.jl: A generic, simple and fast alphazero implementation. `https://github.com/jonathan-laurent/AlphaZero.jl`, 2021.

[92] Tom Silver and Rohan Chitnis. PDDLGym: Gym environments from PDDL problems, 2020.

[93] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[94] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

[95] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[96] Tom Silver, Rohan Chitnis, Aidan Curtis, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Planning with learned object importance in large problem instances using graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11962–11971, 2021.

[97] Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Glib: Efficient exploration for relational model-based reinforcement learning via goal-literal babbling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11782–11791, 2021.

[98] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, pages 167–181. 2014.

[99] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 221–236, 2019.

[100] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[101] Jun S Liu, Rong Chen, and Tanya Logvinenko. A theoretical framework for sequential importance sampling with resampling. In *Sequential Monte Carlo methods in practice*, pages 225–246. Springer, 2001.

[102] Marco F Cusumano-Towner and Vikash K Mansinghka. Using probabilistic programs as proposals. *arXiv preprint arXiv:1801.03612*, 2018.

[103] Caelan Reed Garrett, Chris Paxton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Dieter Fox. Online replanning in belief space for partially observable task and motion problems. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5678–5684. IEEE, 2020.

[104] Rohan Chitnis, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrating human-provided information into belief state representation using dynamic factorization. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3551–3558. IEEE, 2018.

[105] Ronen I Brafman and Guy Shani. Replanning in domains with partial information and sensing actions. *Journal of Artificial Intelligence Research*, 45:565–600, 2012.

[106] Thomas Keller and Malte Helmert. Trial-based heuristic tree search for finite horizon MDPs. In *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[107] Hector Geffner. Heuristics, planning and cognition. *Heuristics, Probability and Causality. A Tribute to Judea Pearl. College Publications*, 2010.

[108] Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

[109] Steven Holtzen, Yibiao Zhao, Tao Gao, Joshua B Tenenbaum, and Song-Chun Zhu. Inferring human intent from video by sampling hierarchical plans. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1489–1496. IEEE, 2016.

[110] Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977*, 2017.

[111] Iris R Seaman, Jan-Willem van de Meent, and David Wingate. Modeling theory of mind for autonomous agents with probabilistic programs. *CoRR*, 2018.

[112] Chris L Baker, Julian Jara-Ettinger, Rebecca Saxe, and Joshua B Tenenbaum. Rational quantitative attribution of beliefs, desires and percepts in human mentalizing. *Nature Human Behaviour*, 1(4):1–10, 2017.

[113] Julian Jara-Ettinger, Laura E Schulz, and Joshua B Tenenbaum. The naive utility calculus as a unified, quantitative framework for action understanding. *Cognitive Psychology*, 123:101334, 2020.

[114] Alwa Alanqary, Gloria Z Lin, Joie Le, Tan Zhi-Xuan, Vikash Mansinghka, and Josh Tenenbaum. Modeling the mistakes of boundedly rational agents within a bayesian theory of mind. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 43, 2021.

[115] Tan Zhi-Xuan, McCoy R. Becker, and Vikash K. Mansinghka. Genify.jl: Transforming Julia into Gen to enable programmable inference. In *Languages For Inference (LAFI) Workshop, 48th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan 2021.

[116] A. Doucet, A. Smith, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice*. Information Science and Statistics. Springer New York, 2013. ISBN 9781475734379. URL `https://books.google.com/books?id=BWPaBwAAQBAJ`.

[117] Dylan Hadfield-Menell, Stuart J Russell, Pieter Abbeel, and Anca Dragan. Cooperative inverse reinforcement learning. *Advances in Neural Information Processing Systems*, 29:3909–3917, 2016.